6502 ASSIST - AN INTERACTIVE SOFTWARE DEVELOPMENT TOOL FOR THE 6502 ASSEMBLY LANGUAGE

by

Victor Wayne Daniel

Bachelor of Science
University of North Carolina, 1965

Doctor of Philosophy
University of Virginia, 1970

Submitted in Partial Fulfillment of the
Requirements for the Degree of Master of Science
in the Department of Computer Science
University of South Carolina
1986

Department of Computer Science
Director of Thesis

Department of Computer Science
Second Reader

College of Engineering
Third Reader

Dean of the Graduate School

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

THE 6502 ASSIST PACKAGE

6502 ASSIST is an interactive program designed to assist students in writing syntactically correct 6502 source code, locating assembly errors, and debugging the source code at run time. 6502 ASSIST is designed primarily for interactive use on a VAX computer system since the VAX file structure is assumed by the ASSIST routines. 6502 ASSIST is written almost entirely in standard FORTRAN 77 except for a few calls to VAX system routines (used to redefine temporarily the function of the Control C and Control Y keys during source code execution). The 6502 ASSIST program can easily be adapted to any computer system supporting FORTRAN 77 provided the VAX system calls are deleted and file access statements are changed to match the host system requirements.

6502 ASSIST consists of three major components: an editor, an assembler and a simulator. A brief description of the function of each as well as special features of each is given in this chapter. A thorough treatment of the internal details of each component is given in chapters 3-5. A user's guide for the entire 6502 ASSIST package appears in the appendix.

1.1 EDITOR

The editor functions like most common line editors where line numbers are assigned to each line. This

facilitates accessing individual lines or a range of lines. Most editor commands permit line number ranges as parameters. The editor supports cut and paste type operations where whole ranges of lines can be moved or copied from one place to another. In addition, the editor can append any existing file to the current file being edited.

A special feature of the editor is its ability to check lines for proper 6502 syntax. This can be done in three ways. First, new lines being inserted from the keyboard can automatically be checked for any syntax errors. If any errors are detected, they are immediately listed and the user is requested to retype the entire line. If this feature is not desired, the user can turn the syntax checking mechanism off via the O editor command. Second, the user can check the syntax of an individual line or a range of lines at any time via the C or K commands. This provides for a more thorough check than the first method since a partial symbol table is constructed for the range of lines being checked. Third, the user can check the entire file for all assembly errors via the C command without any line number parameters. This command invokes the full power of the assembler without having to leave the editor or endure a lengthy assembly listing. The object file is created and any assembly errors detected are listed at the terminal using the editor line numbers. This is the preferred

way go for a quick, but complete, assembly check of the source code before entering the simulator for run time testing. If desired, the assembler or the simulator can be called directly from the editor. This saves retyping the file name since it is assumed the user wishes to assemble or run-time test the current file.

The editor permits special symbols to denote the first line (^), the current line (.), and the last line (*). All commands that permit line number ranges may be used with these symbols as well. As an aid in traversing through the file, the user can list individual lines one at a time in the forward direction by repeatedly typing a carriage return.

The editor has the standard commands for printing, inserting, deleting, and replacing individual lines or a range of lines. Normally lines are numbered in multiples of 100, but there is a command to renumber all the lines in multiples of any (integer) incremental size. There are four substitute commands. Two of these substitute only the first occurrence of the search string on each line; the other two substitute all occurrences. There is a find command. Both the find and substitute commands remember their last search strings. Thus they can be subsequently invoked without any parameters to repeat the previous command. Finally there are commands to save the edited file and exit the editor.

1.2 ASSEMBLER

The assembler is a two pass assembler. The assembler attempts to detect as many errors as possible on each line, rather than stopping error checking on detecting the first error. Thus, for example, if the opcode is invalid, the assembler still evaluates the operand expression to detect any additional errors. The usual action for an address field error is to assume a value of zero, print an error message, and proceed with the assembly. For those errors not so easily patched, the usual action is to print an error message and ignore the statement. For most errors the assembler prints a message indicating the action taken to "correct" the error. If not, one can assume that the action taken is to ignore the statement. In any event, an inspection of the assembly listing will reveal whether any code was generated for the statement in question. Regardless of the default action taken, an error in one statement could generate errors in subsequent statements.

The assembler recognizes all standard 6502 pseudo-operations. In addition, ORG and EQU may be used for the origin and equate pseudo-ops instead of the usual *= and = operators. Labels must start with a letter and contain only letters and digits. The assembler truncates all labels to the first six characters. The label, operation code, and address field must be separated by blanks or tab characters. Embedded blanks are not

permitted within the address field. Arithmetic expressions consisting of numbers and labels are permitted within the address field. Numbers may be expressed in decimal, hexadecimal, octal, or binary. Addresses may be given as an offset from the program counter (*). A single ASCII character preceded by an apostrophe is also allowed as an address. Use of the .END psuedo-op is optional.

The assembler provides an assembly listing in which memory locations and the values stored in them are shown in hexadecimal. The source lines are also shown but they are renumbered in decimal beginning with statement number 1. For those statements that do not generate any code (pseudo-ops or statements with severe errors), a location is shown but no code is shown stored there (indicated by a blank field). The sample assembly listing shown below should clarify most of these concepts.

```
                 ***ASSEMBLY LISTING***

LC          CODE          STMT       SOURCE LINE
0000:                      1         ORG $900
0900:                      2 COUNT   EQU $00
0900:                      3 SUM     EQU $01
0900:       A5 00          4         LDA $0
0902:       85 00          5         STA COUNT
0904:       A2 0A          6         LDX #10
0906:       18             7         CLC
0907:       E6 00          8 LOOP    INC COUNT
0909:       65 00          9         ADC COUNT
090B:       CA            10         DEX
090C:       D0 F9         11         BNE LOOP
090E:       8D 00 00      12         STA SUMM
   ***  SUMM IS AN UNDEFINED SYMBOL - VALUE 0 ASSUMED
0911:       00            13         BRK
   *** UNSUCCESSFUL ASSEMBLY - 1 ERROR(S) ***
```

1.3 SIMULATOR

The simulator is used to run-time test 6502 source code. The simulator simulates a 6502 microprocessor with 64K RAM. In addition, the simulator includes some functions of the 6522 VIA I/O chip. This feature permits simulation of I/O operations during run-time testing of 6502 source code. Five RAM addresses are reserved for I/O ports and data direction registers. These are memory locations 9110-911F. A more thorough treatment of this and other simulator features is found in the User's Guide.

The simulator has extensive debugging facilities. These include commands to print source lines with or without execution count, to print contents of memory in hexadecimal or as ASCII characters, to fill memory with any hex value, or to search memory for any hexadecimal or ASCII string. The heart of the debugger is its five execution modes. In the "go" mode the simulator runs at full speed, stopping only at program breaks (the BRK instruction). In the "quick trace" mode the simulator runs at full speed, but stops at all program breaks, software breakpoints, and non-source instructions. Software breakpoints can be set at any source line (except at ORG/EQU pseudo-ops) via the B command and removed via the RB command. In the "trace" mode, the simulator prints each source line just before it is executed. Like quick trace, trace stops at all program breaks, software breakpoints, and non-source instructions. In the "walk"

mode, the simulator executes the current instruction and lists the next instruction. Typing a carriage return will continue the walk once the W command is issued. If desired, the current values of all registers can be displayed along with the source instruction just before the instruction is executed. This feature applies only to the trace and walk execution modes. The O command is used to turn on/off the register printing switch.

The final execution mode is the "jump" mode. This mode is used to execute all instructions in a subroutine call. The jump mode can be invoked at any time via the J command. If the current instruction is a JSR instruction, then the entire subroutine call is executed. If the current instruction is already within a subroutine call, but is not a JSR instruction, then the remainder of the current subroutine call is executed. Any subroutine calls nested within the current call are also executed. If the current instruction is not a JSR instruction and is not within any subroutine call, then the jump mode acts like the walk mode, executing the current instruction and listing the next one. The jump mode always prints the next instruction to be executed after all instructions in the current subroutine call have been executed. The registers are displayed if the register printing switch is on (controlled by the O command). However, one can view the register values at any time via the R command.

The simulator provides a way to stop program

execution while in progress. This is done by typing a Control C character. The simulator prompt "." will appear and any simulator command may be given at this point. Typing a carriage return will continue program execution.

The simulator also provides a way to signal interrupt requests during program execution. This feature is useful for testing interrupts routines as well as source program handling of interrupt requests. Typing a Control Y character during source code execution signals an interrupt request. Multiple interrupt requests are permitted. The trace mode is probably the most practical execution mode for initial testing of interrupt requests since the trace mode prints each statement as it is being executed. Also the trace mode is fairly slow due to its printing aspects (the speed of the trace mode is largely controlled by the baud rate of the user's terminal). Trapping the Control Y and Control C characters during program execution requires the use of VAX system routines.

Other useful debugging features are commands to change the contents of any register, to display the current values of the I/O ports and data direction registers, to set the statement limit (default limit is 10000), and to reset the simulator. The reset command zeros out all registers and counters. A particularly useful command for locating errors when a program does not terminate as expected is the dump command. The dump command prints the last 10 source instructions executed

and the last 10 branch instructions executed. The simulator keeps track of the number of times each statement is executed. This information can be very valuable in locating run time errors. The K command is used to print each source statement together with its execution count.

There are two commands for matching source statements to memory locations. The A command shows where in RAM the code for a given range of source lines is stored. The S command shows which source statements correspond to a given range of RAM locations. This information can also be gleaned from a hardcopy of the assembly listing. In addition, there is a command that disassembles the code stored in a given range of RAM. The disassembly is in a pure form with addresses in hexadecimal (no labels) and is independent of the source statement listing.

The simulator can load additional source files via the L command. While loading new source statements from a file, the simulator checks for overwriting of source code in RAM. If the code for any source statement overwrites the code for a previously loaded source statement (whether from the current file or a previous file), an appropriate warning message is printed and the statement is loaded anyway. This is a useful feature when multiple source files must be loaded before run-time testing can begin.

Many simulator commands permit statement number parameters or memory location parameters. Where

appropriate, ranges are permitted as parameters. For example P5,8 prints source lines 5 to 8 while M1C40,1C90 prints memory locations 1C40 to 1C90. Note that line number parameters must always be given in decimal form while memory location parameters must always be given in hexidecimal form. The simulator also permits special symbols to denote the first line or the first memory location (^), the current line or the current memory location (.), and the last line or the last memory location (*). The first memory location is always 0000 and the last memory location is always FFFF. Note that the current line may or may not correspond to the current memory location, depending on the previous commands issued. These special symbols may be used anywhere line number or memory location parameters are permitted. Their use in the simulator is identical to their use in the editor.

The carriage return and backslash keys can be used to traverse through the source file or through memory in a forward direction or in a backward direction. For forward motion repeatedly press the carriage return key. For backward motion, repeatedly press the backslash and carriage return keys in succession. These keys function in this manner for all simulator commands for which forward motion or backward motion make sense (for example, printing source lines or memory locations). For those commands where only forward motion makes sense (for

example, the program execution commands), the backslash and carriage return keys operate only in forward direction.

The B command sets software breakpoints. Software breakpoints are not source code and should not be confused with program breaks (the BRK instruction). The simulator stops at all software breakpoints during program execution via the Q (quick trace) or the T (trace) commands. Software breakpoints are inserted at critical locations in the source code for debugging purposes. The B command can be used to set breakpoints that are recognized only after a specified number of executions have occurred. The RB command removes individual breakpoints or a range of breakpoints. The VB command is used to verify (list) all currently active breakpoints. Note that the quick trace and trace execution modes are particularly useful for run-time testing of programs that "run wild" through memory since these execution modes stop at all nonsource instructions as well as all software breakpoints. In other words, these modes keep tight control over the exection process. Finally, there is a command to exit the simulator.

CHAPTER 2

HISTORICAL OVERVIEW

In early 1982 when the 6502 project was still in its infancy, a search of the literature revealed there were no comprehensive 6502 software development tools available. About all that were found were a few unrelated emulators, cross assemblers, and one simulator. Even now after four years have elapsed, there are no 6502 packages on the market other than 6502 ASSIST that combine the advantages of a 6502 syntax checking editor, a 6502 assembler, and a 6502 simulator into one coherent product. A recent literature search revealed, to be sure, more emulators, cross assemblers, tracers, and simulators on the market today, but no packages that combine the features of any two or more of these products. In fact, there appear to be no other 6502 syntax checking editors at all.

2.1 NEED AT MONTANA TECH

In the spring of 1980 the first microprocessor course was offered at Montana Tech. In those days the students had to assemble by hand their 6502 programs before entering them into the KIM microcomputers. By 1982 the college had purchased several VIC 20 personal computers with the VICMON extension ROM. Even then students often would hand assemble their 6502 programs before coming to the lab sessions (to save valuable lab time for debugging and testing programs). The VICMON

monitor did include a weak assembler that was capable of assembling a single instruction at a time provided it contained no forward branches or labels. There were few debugging aids to help students locate obscure programming errors. Outside of the laboratory there were no debugging or testing facilities at all.

6502 ASSIST became fully operational in mid 1983. After this date students in the microprocessor course could assemble, debug, and run their 6502 programs outside of the laboratory session using 6502 ASSIST. All a student needed was an account on the central computer system (VAX 780's) to access 6502 ASSIST from any of the 200+ workstations on campus. According to the instructor for the microprocessor course, students now come to the laboratory sessions better prepared with programs that are essentially debugged, except for occasional hardware interface problems. One student has written a program to download the 6502 object files created by 6502 ASSIST directly into the VIC 20 memory, thus eliminating all data entry errors. This allows students to burn EPROM's from 6502 software previously tested on 6502 ASSIST without error.

6502 ASSIST has played a major role in the development of a mine disaster robot called MONTY by a group of Montana Tech faculty and students. MONTY's task is to investigate mine tunnels that are not safe for human

passage. Such a condition usually occurs after some type of mine disaster. All of MONTY's software was developed and tested using 6502 ASSIST before field tests began. After thorough field tests, EPROM's were burned and placed in MONTY'S memory banks. As MONTY continues to evolve, 6502 ASSIST is expected to continue to be the major software development tool for the MONTY project.

What is clear is that 6502 ASSIST is a reliable product that has been thoroughly tested at Montana Tech for more than two years now. Surprisingly, since the first month of its operation, there have been no errors reported for this product.

2.2 INNOVATIVE FEATURES

6502 ASSIST contains some features found in no other product, such as a 6502 syntax checking editor and a simulator that traces using the original source code. What is most remarkable is that such a large assortment of 6502 software development tools can be found together in any one package. Primary features of 6502 ASSIST are the following:

1) A user friendly, interactive environment where all messages are written in plain English,
2) A general purpose editor that can perform three levels of 6502 syntax checking,
3) An assembler that supports the full set of MOS technology conventions and detects as many errors

as possible on each pass,

4) A simulator that has extensive debugging facilities, including four execution modes,

5) Full 64K of simulated RAM memory,

6) The same command structure for all components and ease of mobility between the components,

7) Program tracing displaying original source code,

8) Simulated I/O ports that permit dynamic I/O operations,

9) Ability to dynamically signal interrupts during source code execution,

10) Ability to dynamically halt program execution at any time, examine and modify memory, and restart program execution, and

11) Editor and simulator commands similar to those used in software with which the students are already familiar, namely the VAX/SOS editor and the VICMON monitor.

2.3 COMPARISONS WITH OTHER PRODUCTS

An extensive literature search of three high technology databases (INSPEC, Computer Database, and Microcomputer Index) yielded several 6502 software development tools that have been marketed, sometimes for a nominal cost, during the last five years. Mostly these products consist of assemblers, simulators, emulators, and debuggers. Some are resident software, others are cross

software. No 6502 syntax checking editors were discovered, however.

A small number of cross assemblers have been developed. Zeck (1979) developed a simple cross assembler for the University of Washington that runs under UNIX on a DEC PDP-11/60. A more interesting cross assembler is available from 2500 AD Software Inc. (Lloyd, 1984). This cross assembler runs under CP/M on a Z80 microprocessor. It produces either relocatable or absolute code, supports macros, and includes a linker and a loader. It also has an option to translate Z80 syntax directly into 6502 object code. Unfortunately, this cross assembler is extremely memory hungry and sometimes overwrites the CP/M operating system. Needless to say, the system crashes when this occurs. A third cross assembler is available from Sorcism (Freiberger, 1981) which actually can assemble either Z80, 6502, 6800 syntax. Although the 6502 ASSIST assembler does not handle macros, it does support the full set of MOS technology conventions. Some of the weaker assemblers do not. A strong feature of the 6502 ASSIST assembler is its diagnostics. The assembler attempts to detect as many errors as possible on each line of syntax and all error messages are written in plain English.

The literature search revealed several simulators, emulators, and debuggers on the market. The first simulator developed appears to be the KSIM simulator, a

product of Nagata and Miller (1981) at the University of Washington. KSIM runs under UNIX on a DEC PDP-11/60 and is written entirely in C. Like the 6502 ASSIST simulator, KSIM is interactive and permits the user to set breakpoints, examine and alter registers and memory, and trace program execution. KSIM also provides an on-demand menu of simulator commands and allows for a temporary escape to the UNIX monitor. However, KSIM has several limitations that 6502 ASSIST does not have. KSIM has only one execution mode - full speed. The tracing feature is not interactive and does not display the original source code. Furthermore, there are only 3K bytes of simulated memory and there are no interrupt capabilities (in the original version). KSIM can load only one object file and that file must be loaded starting at address 0000. Like 6502 ASSIST, KSIM keeps track of the number of instructions executed and stops execution when the maximum statement limit is exceeded. KSIM also maintains a time limit.

Another simulator is the Visible Computer (Little, 1983) which is a resident simulator for the Apple personal computer. This simulator takes advantage of the Apple's graphics capabilities. The 6502 CPU with its registers are displayed graphically. Animated hex digits dart across the screen to show data transfers to and from memory. Data in the registers and memory can be displayed

in binary, hexadecimal, or decimal notation. Object code can be executed at four levels of detail. The lowest level is the microlevel where the individual steps within the execution of a single instruction are displayed. The highest level is full speed execution by the actual processor (no simulation). Like 6502 ASSIST, the simulator permits the user to examine and modify memory, disassemble a range of memory, and load multiple object files. However, there is no program tracing using the original source code. Although the graphics displays make this an interesting simulator, they do not make up for what appears to be a serious shortage of useful debugging tools.

Two resident debuggers that run on any 6502 based system have appeared in the literature. The first is the 6502 TRACER (Ruppert, 1984) which requires about 1/2K of memory plus two bytes of page zero memory. 6502 TRACER provides a program trace displaying the raw object code and the register values as each instruction is executed. 6502 TRACER does not disassemble code nor collect program statistics, such as the number of times each instruction is executed. There are no interrupt capabilities. Obviously 6502 TRACER is a weak debugging tool. A somewhat better resident debugger is the CONTROL progam (Esformes, 1984). CONTROL requires about 1K of memory plus 4 bytes of page zero memory. CONTROL provides for single step tracing with disassembly of each instruction.

For full speed execution the user can set breakpoints at specified instructions or memory locations. CONTROL can be programmed by the user to perform special operations before each instruction is executed. For example, CONTROL can be programmed to collect program statistics such as the number of machine cycles used. Although 6502 ASSIST collects program statistics without special programming, the programmable feature of CONTROL does provide for more flexibility in what statistics can be collected. The only serious drawback is the effort required to program CONTROL to perform the tasks desired by the user. CONTROL is not a debugger for the novice.

Da-Tech (Wittenberg, 1984) markets a standalone in-circuit emulator that allows real time operation of the user system without hardware or software modification. The emulator runs on a separate 6502 microprocessor. The unit permits the user to display and alter registers and memory, halt a program at a specified address, step through a program one instruction at a time and stop a program at a location after a number of loops have been completed. Of course, 6502 ASSIST can do all of these things and more.

6502 ASSIST is named after the popular ASSIST assembler/interpreter for the IBM 360/370 instruction set. This non-6502 based product was developed in the late sixties by Mashey and Campbell (1974) at Pennsylvania

State University. 6502 ASSIST incorporates many ASSIST features. For example, both packages provide for a program dump of the last 10 instructions executed and the last 10 branch instructions executed. Both aid debugging through excellent diagnostics. Both monitor the amount of resources used and terminate program execution if it is excessive. Both monitor the number of instructions executed. However, ASSIST also monitors the amount of time used and the amount of output produced. Both provide summary statistics. 6502 ASSIST even provides for an execution count for each source instruction. Both provide for program tracing. ASSIST supplies some extra conveniences such as its macro facility and its special psuedo-commands to facilitate I/O operations and number conversions. Being interactive, 6502 ASSIST has one clear advantage - it creates a friendly environment in which the user feels in control at all times.

CHAPTER 3

THE EDITOR - HOW IT WORKS

The editor is a line oriented syntax checking editor. Each line is assigned a line number to facilitate access to that line. Optionally, each new line being entered under the control of the editor may be checked for proper 6502 syntax as soon as the line is typed.

3.1 DATA STRUCTURES

The editor stores source lines in a large array X declared as CHARACTER*80 X(999). The choice of 999 as the array size is based solely on the fact that lines are normally numbered in multiples of 100 and the editor displays 5 digit line numbers (thus lines 00100 to 99900 would fill the array X).

In order to avoid moving lines of text stored in array X, two other arrays are used. Array LN stores the line numbers and array LINK stores the corresponding locations of those lines in array X. Figure 1 illustrates the relation between these three arrays.

From this example, we can see that deleted lines remain in array X until the next editor session. This should present no problem since array X is fairly large. In the event that space becomes a problem due to a large number of deletions, one can simply exit the editor (thereby saving the file) and re-enter the editor (thereby getting a clean copy of the file).

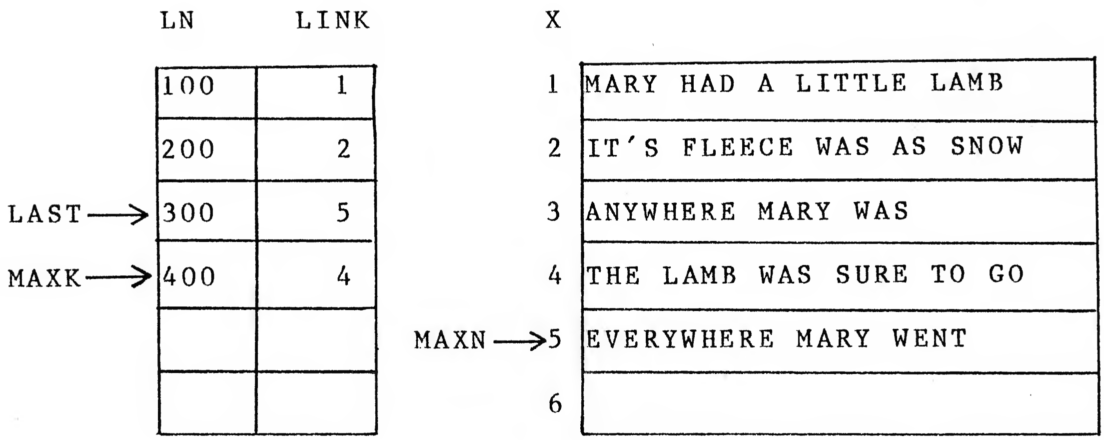

Figure 1. Editor Data Structures

The line numbers stored in array LN always remain sorted by increasing line number. Thus when new lines are inserted between existing lines, entries in LN (and LINK) are moved downward to make room for the new line numbers. However, the new lines themselves are stored at the next free locations in array X. When lines are deleted, their numbers are removed from array LN. Thus entries in LN (and LINK) are moved upward to remove the gaps. The deleted lines themselves remain in array X, but they become inaccessible. The editor maintains three pointers to facilitate line manipulations:

LAST - points to last line number accessed in array LN

MAXK - points to last line number stored in array LN

MAXN - points to last line stored in array X

The pointer LAST is used to access the current line, while pointer MAXK is used to access the line at the physical end of the edited file. The pointer MAXN is used

to keep track of where to insert new lines in array X. These pointers are illustrated in figure 1. Most of the editor commands are performed in a straight forward manner using these three pointers and the three arrays LN, LINK, and X.

3.2 COMMAND LEVEL

The top level logic of the editor is contained in module EDIT. EDIT begins by interrogating the user for a file name and the status of the file (old or new). If the file is an old file, the file is read into array X with line numbers stored in array LN. The lines are numbered in increments of 100 with line 100 being the first line. EDIT also asks the user if he wishes new lines being inserted into the file to be automatically checked for proper 6502 syntax. If he answers affirmatively, then the syntax checking switch CHECK is turned on (CHECK=.TRUE.), otherwise CHECK is turned off (CHECK=.FALSE.). The O command can be given at any time to reverse the logic value of CHECK, thus turning the syntax checking switch on if it was off or off if it was on.

After the opening formalities are over, control in EDIT passes to a DO FOREVER loop in which commands are read at the top of the loop and compared with each of the 23 valid commands, if necessary, until a match is found. When the linear search produces a match, the appropriate modules are called to implement the command.

If no match is found, then a warning message is printed at the terminal. The loop is then repeated indefinitely until the E (exit) command is given.

Many editor commands permit line number ranges to be specified. For these commands, EDIT calls module READ to read the remainder of the command line for any ranges. For example, P500,800 would be interpreted as the print command to print all lines in the range 500 to 800.

3.3 SUPPORTING MODULES

Module PRINT is used by the editor to print a range of lines. PRINT calls module PRNT to print a single line. PRNT trims trailing blanks from each line in order to speed up printing at the terminal.

The insert command calls module INSERT to insert new lines between existing lines or at the end of the file. The insert command permits a beginning line number and a line number increment to be specified. If the line number increment is omitted, the normal action is to insert the first new line at the specified location and use a default increment of 100. Insertion automatically stops when any incremented line number exceeds the line number of the following line (if there is one). The insertion mode can be terminated at any time by typing a Control Z character.

The replace command calls module REPLAC to replace individual lines or a range of lines. REPLAC overwrites

the given range of lines in array X with the new lines. The old line numbers are retained. It is possible to replace the old lines with fewer new lines (the Control Z character must be used to signal early termination). In this case, any remaining old lines in the range being replaced become inaccessible (hence effectively deleted). When the syntax checking switch is on, both REPLAC and INSERT call module ECHECK to check the syntax of new lines being inserted. Module ECHECK will be discussed in more detail shortly.

The exit and the write commands call module WRITE to save the source lines in an external file. WRITE uses the file name specified at the beginning of the editor session. If the file was an old file, a new version is created on VAX VMS systems. This module might have to be modified somewhat for use on other computers systems since most systems do not permit multiple versions of a file with the same file name.

The delete command calls module DELETE to delete a range of lines. As mentioned earlier, the lines themselves are not removed from array X, but they become inaccessible for the remainder of the editor session.

Module APPEND is used to append an existing file at the end of the current file. The appended lines are inserted at the next available locations in array X and are given line numbers higher than those lines already

stored in X. If there is insufficient space in array X, a warning message is printed and the file is not appended.

The editor has two modules, ECHECK and LCHECK, for checking lines for proper 6502 syntax. When the syntax checking switch is turned on, ECHECK is automatically invoked to check new lines being inserted by the insert or replace commands. The editor commands C and K call module LCHECK to check a range of lines. The syntax checking performed by LCHECK is somewhat more thorough than that done by ECHECK in that LCHECK uses the values of any labels previously computed to evaluate subsequent expressions. Thus LCHECK can often detect addresses out of range whereas ECHECK, which looks at only one line of code, might not. Essentially, LCHECK performs pass one of the assembler on a range of lines. However, LCHECK performs both passes whenever the C or K commands are given without any parameters. In this case, the entire file is assembled and the object file is created, just as with the assembly command (the difference being that the C and K commands assemble using the editor line numbers for error diagnostics whereas the assembler uses its own line numbers). Both ECHECK and LCHECK call the assembler routine SCHECK to check the syntax of individual lines and build the symbol table.

All four of the substitute commands and the find command are performed by the module SUBST. SUBST uses the intrinsic function INDEX, standard in FORTRAN 77, to

locate substrings to be replaced. The find command calls module SUBST to replace the search string with itself. SUBST must move the tail end of any line in which a substitution occurs in which the replacement string is longer or shorter than the search string.

The move and copy commands are performed by module MOVE. This module automatically assigns new line numbers for the inserted text based on even spacing in the range provided. This usually leaves sufficient space for further insertions between the relocated lines. The copy command stores a duplicate copy of the given lines in array X, while the move command only manipulates the LN and LINK arrays to reflect the new line numbers.

The editor can call the assembler directly (module ASSEMB) via the A command if an assembly listing is desired. If there are no assembly errors, the user can then call the simulator directly (module EXECUT) from the editor via the X command. The only advantage to this procedure is one of convenience to the user since he does not have to specify the file name again (the file name is passed automatically to the assembler or the simulator).

CHAPTER 4

THE ASSEMBLER-HOW IT WORKS

The assembler is a two pass assembler. The primary module for both passes is SCHECK. SCHECK checks an individual line for proper 6502 snytax (both passes), builds the symbol table (pass one), evaluates the operand expression (both passes), performs any psuedo-operations (usually pass two), and writes the assembly code for the line to the object file (pass two). The assembler (SCHECK) attempts to find as many errors as possible on pass one. This is done so that SCHECK can be used by the editor for a thorough one pass syntax check of individual lines or a range of lines. This results is some redundancy, such as evaluating operand expressions on both passes when often they are not needed until pass two. Sometimes operand expressions can't be evaluated on pass one due to the presence of undefined symbols. In this case, the expression is ignored until pass two.

4.1 DATA STRUCTURES

The assembler uses four fixed tables (arrays) to encode instructions: MOT (machine operation table), OPCD (opcode), MD(mode), and LEN (length). These are declared as follows:

```
CHARACTER*3     MOT(112),OPCD(0:255)
INTEGER         MD(0:255),LEN(0:13)
```

MOT stores the 56 opcode mnemonics in both uppercase and lowercase. MOT is searched in both passes to determine if the mnemonic is valid. If it is, then it is converted to uppercase (if not already) before further processing.

OPCD stores the opcode mnemonics in uppercase only for each of the 256 possible opcodes (only 151 opcodes are valid). The companion array MD stores the addressing modes for each of the 256 possible opcodes. These two arrays are searched jointly to determine if a particular instruction is valid; if so, the assembler has found the correct opcode. The following integer codes are used to represent the 13 valid addressing modes:

0 - Invalid

1 - Immediate

2 - Zero page, direct

3 - Zero page, indexed on X

4 - Zero page, indexed on Y

5 - Pre-indexed, indirect

6 - Post-indexed, indirect

7 - Relative

8 - Absolute, direct

9 - Indirect

10 - Absolute, indexed on X

11 - Absolute, indexed on Y

12 - Accumulator

13 - Implied

LEN stores the lengths (in bytes) of the instruction for each of the addressing modes. The invalid addressing mode 0 is assigned a length of 0 bytes. The three arrays OPCD, MD, and LEN are also used in the simulator to disassemble raw instruction codes.

The assembler maintains two arrays ST1 and ST2 for storing labels (ST1) and their values (ST2). Jointly these form the symbol table. Up to 200 entries are permitted in the symbol table.

The final table maintained by the assembler is MT (memory table). MT is used by the assembler to store the statement numbers of those instructions which on pass one could be either two byte or three byte instructions, depending on the values of their address fields. This ambiguity only comes up when undefined names appear in the address field during pass one for certain type addressing modes (absolute/zero page, direct/indexed). Since it is impossible to tell which length is correct on pass one, a three byte format is assumed for these type instructions. The assembler on pass two checks table MT to see if the current statement number is there. If so, the three byte format is enforced regardless of the value of the operand field.

4.2 FIRST PASS

On pass one, the assembler reads an individual line from the source file. SCHECK is called to check the

syntax of the source line and to determine the length of the instruction. SCHECK calls BREAK to break the line into a label, an opcode mnemonic, and an address field. SCHECK calls CHECK to check that the label starts with a letter, contains only letters and digits, and is not a duplicate label. Next SCHECK searches to see if the opcode mnemonic is a pseudo-op. If so, it evaluates the operand expression (mostly to generate any error messages), changes the location counter for ORG/EQU psuedo-ops, and terminates pass one for an END pseudo-op. If the opcode mnemonic is not a pseudo-op, SCHECK calls function INDX to return the index of the mnemonic in table MOT. INDX returns 0 if the mnemonic is not found (invalid). SCHECK then calls SCAN to scan the address field to determine the address mode and instruction length. SCAN also stores in array MT the line number for any instructions for which the instruction length is ambiguous (thereafter assumed to be three bytes in length). Next SCHECK calls function HEXCD to convert the mnemonic/address pair into the proper opcode. If the address mode is invalid for the given opcode mnemonic, a value of 22 hex is returned (an invalid opcode). Finally SCHECK inserts any non-blank label, if valid, into the symbol table. After SCHECK completes its tasks, the location counter is incremented by the instruction length. The assembler repeats the above process until all source lines have been read or the END pseudo-op is encountered.

4.3 SECOND PASS

On pass two, the assembler rewinds the source file before reading individual lines. As in pass one, SCHECK is called upon to syntax check individual lines. SCHECK calls BREAK to decompose the line into its three fields. Labels are not checked in pass two. Next SCHECK searches to see if the opcode mnemonic is a pseudo-op. If so, it performs the required actions such as writing data to the object file for BYTE, DBYTE, WORD, and TEXT pseudo-ops, changing the value of the location counter for an ORG pseudo-op, or terminating pass two for an END pseudo-op. If the mnemonic is not a pseudo-op, SCHECK calls function INDX to return its index in table MOT. SCHECK then calls SCAN to determine address mode and instruction length. SCAN checks array MT before determining instruction length for certain address modes (any ambiguous cases from pass one were stored in MT). Next SCHECK calls function HEXCD to convert the mnemonic/address pair into a proper opcode. If an invalid opcode is returned, the instruction length is set to 0 and the instruction is effectively ignored. Finally SCHECK assembles the parts of the instruction and writes the object code for the line to the object file. After SCHECK completes its tasks on the given line, the location counter is incremented by the instruction length. The assembler repeats the above process until all source lines have been assembled or the END pseudo-op is

encountered. Finally the assembler calls LIST to print an assembly listing indicating all errors found.

4.4 SUPPORTING MODULES

There are several minor routines that are used by the assembler, but have not been mentioned so far. These are VALUE, OPRAND, VAL, CH2, CH4, HVAL, OVAL, and BVAL. The most important of these is function VALUE. VALUE is called often to compute the value of the address field, which can be any integer expression using numbers and names as operands and +, -, *, and / as operators (no parenthesis or embedded blanks permitted). VALUE calls OPRAND to pick off the operands in the expression and function VAL to evaluate each operand. VAL in turn calls HVAL, OVAL, or BVAL to compute the integer value of hexadecimal, octal or binary operands, respectively. Finally, CH2 and CH4 are used to convert integer values to hexadecimal values of lengths 2 and 4 characters, respectively. CH2 and CH4 are used to convert the instruction codes to hexadecimal just before writing them to the object file.

The assembler uses several files. Two of these have already been mentioned - the source file on unit 1 and the object file on unit 2. In addition to these, the assembler uses several temporary or 'scratch' files to store error messages on each pass. The scratch file on unit 4 stores all error messages on pass one. The scratch file on unit 8 stores all error messages on pass two.

Since some error messages would be identical for a given line on both passes, the file on unit 9 stores error messages for all pass one errors that could not be detected by pass two. An example of this would be an undefined name in an ORG statement on pass one, but later given a value before pass one completes. Pass two could not detect this type of error. The scratch file on unit 4 is used solely for listing the pass one errors detected by the editor syntax checking commands C and K. The assembler itself uses only the error messages stored on the unit 8 and unit 9 files. Together these represent all the errors detected by the assembler on both passes.

CHAPTER 5

THE SIMULATOR - HOW IT WORKS

The simulator simulates a 6502 microprocessor with 64K RAM. In addition, the simulator has extensive debugging facilities. The debugging facilities require considerably more memory than a simple 64K simulator.

5.1 DATA STRUCTURES

Five large arrays are used by the simulator: MEM (memory), MS (memory source), SLINE (source line), COUNT, and LOK (location). These are declared as follows:

```
CHARACTER*40 SLINE(999)
INTEGER   COUNT(999), LOK(999)
INTEGER   MEM(0:65535), MS(0:65535)
```

SLINE stores the source lines created by the 6502 syntax checking editor or any other editor. Note that a shorter line width (40 characters) is used in the simulator. It is assumed that the majority of 6502 programmers will not want to write lines of assembly code longer than 40 characters - the typical screen width used in 6502 based systems. COUNT stores the execution count for each source line. The execution count is the number of times the code for the source statement is executed by the simulator during a run of the code. LOK stores the location in memory where the code for each source line is stored. To be more precise, LOK(N) is the address of the

first byte in memory (array MEM) where the assembly code for line N is stored.

Array MEM simulates the 64K RAM memory. MEM stores source code as well as data. The debugging facilities require a second 64K array MS to keep track of the locations of the source lines in SLINE corresponding to the source code stored in MEM. Thus the debugger uses LOK to find the source code location given the line number and MS to find the line number given the source code location. These arrays make it possible for the debugger to alert the programmer to events such as control transferring out of the source code or events that change the source code but not the source lines. These are extremely useful debugging tools for programs that run wild through memory.

To illustrate the use of these five arrays, consider the following short 6502 program:

```
ORG $5
LDA #$20
STA $1
LDA #$30
STA $2
CLC
ADC $1
STA $3
BRK
```

After this program is assembled and executed, the five simulator arrays would appear as shown in figure 2. Note that the ORG/EQU pseudo-ops do not generate any code to be stored in memory (array MEM). A location value of -1 is stored in LOK for such source lines. Also note that the values stored in arrays LOK and MEM above are

shown in hexadecimal for convenience. Whenever values in these two arrays are displayed via debugger commands, the displayed values are always in hexadecimal form (except for the I and IW commands).

	SLINE	COUNT	LOK
LASTS → 1	ORG $5	0	-1
2	LDA #$20	1	5
3	STA $1	1	7
4	LDA #$30	1	9
5	STA $2	1	B
6	CLC	1	D
7	ADC $1	1	E
8	STA $3	1	10
MAXS → 9	BRK	0	12
10			
11			
12			

	MEM	MS
0	00	0
1	20	0
2	30	0
3	50	0
4	00	0
5	A9	2
6	20	0
7	85	3
8	01	0
9	A9	4
A	30	0
B	85	5
C	02	0
D	18	6
E	65	7
F	01	0
10	85	8
11	03	0
LASTM → 12	00	9
13	00	0

Figure 2. Simulator Data Structures

The simulator maintains two pointers to facilitate source line listings and one pointer to facilitate memory manipulations:

LASTS - points to last line accessed in array SLINE

MAXS - points to last line stored in array SLINE

LASTM - points to last location accessed in array MEM

The pointer LASTS is used to access the current source line, while the pointer MAXS is used to access the source line at the physical end of the source file. The pointer LASTM is used to access the current location in memory, which may or may not correspond to the current source line. Some debugger commands such as the P command only move the current line pointer LASTS, other debugger commands such as the M command only move the current memory location pointer LASTM, while still other commands such as the S command move both pointers.

Most of the simulator commands are performed in a fairly straightforward way using the five arrays SLINE, COUNT, LOK, MEM, and MS and the three pointers LASTS, MAXS, and LASTM. One exception to this is the way breakpoints are stored in array MS. If in the preceding example, we wished to set a breakpoint at source line 4 via the B command, the value stored in MS(9) would be changed from 4 to -4. Thereafter the simulator would break at line 4 everytime the code for line 4 is about to be executed in memory. On the other hand, we could use

last 10 entries are kept in the queues. Thus, there is no need to maintain a pointer to the front of the queue. Initially both queues are zeroed out, since there is no line number 0.

Module PUSHQ is used to insert an instruction in the SQ queue. If the instruction is a branch instruction, PUSHQ also inserts it into the BQ queue. Module DUMP is used to print the last 10 source instructions executed and the last 10 branch instructions executed. DUMP simply refers to SQ and BQ to find the line numbers of these instructions and proceeds to print them in the correct order (from the front to the rear of the queue).

5.2 COMMAND LEVEL

The top level logic is contained in module EXECUT. EXECUT begins by initializing all the registers, pointers, and counters to zero except for the stack pointer SP which is set to FF (an empty stack) and MAXCNT which is set to 10000. MAXCNT is the upper limit on the number of instructions that can be executed. MAXCNT can be changed at any time by the N command. EXECUT also zeros out memory arrays MEM and MS. Next EXECUT calls module LOAD to load a source file. The remainder of EXECUT is a rather large DO FOREVER loop in which each command is read at the top of the loop and compared with a predefined list of valid commands. Usually the type of command is determined by its first character. The

the B command to set a break at Line 4, but only after 25 executions of line 4. Then MS(9) would be changed to -25004. From this example we can see that the simulator must always test for a negative value stored in MS(9) before executing the code for line 4 stored at MEM(9). If the value stored in MS(9) is negative, then the simulator changes its sign, divides by 1000 to get the remainder 4 and quotient 25. In this way the source line number 4 and the minimum execution count 25 are both recovered. The simulator then checks the value of COUNT(4) to see if 25 executions have already occurred, if so the simulator halts execution at line 4. The use of array MS to store breakpoints as well as source line numbers saves considerable memory. If we wished to cancel the breakpoint at line 4, we could use the RB command. The simulator would then set the value stored in MS(9) back to 4.

The simulator uses two small arrays SQ (Source Queue) and BQ (Branch Queue) to store the last 10 source instructions executed and the last 10 branch instructions executed, respectively. SQ and BQ in reality only store the instruction line numbers. They are declared as

```
INTEGER SQ(10), BQ(10)
```

Two pointers, LASTSQ and LASTBQ, are used to keep track of the position of the last entry in each table. SQ and BQ are managed somewhat like queues, except that only the

remainder of the command, if any, usually determines the range of the command. For example, P10,20 prints lines 10 to 20 while M00,FF lists the first page of memory. Two special modules are used to read the range part of the command: IREAD and HREAD. IREAD is used for ranges containing (decimal) integer values, while HREAD is used for ranges containing hexadecimal values. Once EXECUT matches the command with a command in the predefined list of valid commands, the appropriate modules are called to implement the command. If the command is illegal or its syntax is illegal, a warning message is printed. The DO FOREVER loop continues to prompt for simulator commands until the E (Exit) command is issued.

5.3 SUPPORTING MODULES

There are 40 or more modules used by the simulator. A few of these have already been mentioned. Next we take a brief look at the more important of these modules and their functions.

Module LOAD is used to load a source file into the simulator. Several source files can be loaded successively by repeated use of the L command. The L command is the only simulator command that can change entries in SLINE, and only by appending new source lines to those already stored in SLINE. For any other changes to the source lines, it is necessary to exit the simulator, use the editor to perform the changes on

the source files, reassemble the source files, reenter the simulator, and reload the source files. Of course, only those source files that are actually changed need be reassembled. The LOAD module, when loading new source lines, always checks to see if the code for each new source line will overwrite the code for any previous source line. If so, the code is loaded anyway, but a warning message is printed on the terminal. The simulator uses array MS to determine if overwriting will occur (recall that MS(K) stores a non-zero value if and only if the code for a source line starts at memory location K). Module LOAD will abort loading a file if there is insufficient space in array SLINE. The simulator can handle up to 999 source lines.

Module TRACE is the heart of the simulator. TRACE is used to execute code stored in memory. TRACE can be operated nominally in 16 different different modes, although only five modes are used by the simulator. This is possible since TRACE incorporates four switches in its argument list that must be set before TRACE is called. These four switches and their functions (when switched on) are listed below:

STPBRK - stop execution at any software breakpoints or non-source instructions

PRTSRC - print source lines corresponding to code being executed

JUMP - jump through current subroutine call (including any subroutine calls nested within)

WALK - execute one instruction

Normally, JUMP and WALK would not both be switched on at the same time since the outcome would be a walk mode. So in reality, TRACE can be operated effectively in only 12 different modes.

TRACE is the most complicated module in the simulator. TRACE calls 14 other modules to assist in executing code and debugging programs. TRACE uses array OPCD to determine the opcode mnemonic from the known opcode. The mode is determined by using array MD in the same manner. Array LEN is then used to obtain the instruction length from the mode. Module LOC is called to determine the memory location of the effective operand from the current PC (Program Counter) value and the instruction mode. Module XINSTR is called to execute one instruction. XINSTR performs a linear search to match the opcode mnemonic with one of 56 possibilities. Once a match is found, the correct action is taken on the effective operand. After the instruction is executed, TRACE calls module PUSHQ to save the instruction temporarily on the queue SQ (used by module DUMP to dump the last 10 instructions executed). TRACE also increments counter NUMX, which counts the number of instructions executed.

Module TRACE, when in jump mode, executes all instruction within the current subroutine call before stopping execution. TRACE uses counter LEVEL to keep track of current depth of subroutine nesting.

Module XPRINT is used for listing the source lines stored in array SLINE. XPRINT calls routine KPRNT to print an individual source line with execution count or routine XPRNT to print an individual source line without execution count. Module MPRINT is used to list code in memory (array MEM) in either hex notation or as ASCII characters. MPRINT lists 16 bytes of memory per line. Similar to MPRINT is module WPRINT which is used to print 1 byte of memory per line. Module TRACE calls XPRNT to print source lines being executed or RPRNT to print source lines with current register values as they are being executed. Whenever TRACE attempts to execute code that does not correspond to the source lines, modules NPRNT or IPRNT are called. NPRNT list non-source instructions and IPRNT lists invalid "instructions".

Module GETADR is used for displaying the memory addresses of a given range of source lines. Module GETSRC is used for displaying the source lines corresponding to a given range of memory addresses.

Module SETBRK is used to set breakpoints at any source line, while module RMVBRK is used to remove breakpoints from any range of source lines. Module GETBRK is used to list all current breakpoints.

Module FILL is used to fill each byte of memory in a given range with the same hexadecimal value. The hex value must be between 00 and FF inclusive. FILL takes the precaution of setting to zero the corresponding entries in array MS to reflect the fact that non-source code is now stored in the given memory range. FILL even searches the two bytes in front of the given range to check whether any part of a 2 byte or 3 byte source instruction will be overwritten by the fill command. If so, the MS entry for that instruction is also zeroed out. FILL also changes the corresponding entries in array LOK to -1 to indicate that the code for these source lines is no longer stored in memory.

Module HUNT is used to search a given range of memory for the occurrence of a group of byte values. The byte values may be given in hexadecimal or as a string of ASCII characters (delimited by a pair of single quote characters).

Module CHANGE is used to change the values stored in any register. The PC register can be altered bit by bit or all the bits at once (a single byte). To perform this task, CHANGE calls module GBITS to break a byte into its component bits and module GBYTE to assemble any 8 bits into a single byte. The 8 component bits of PC are stored independently in variables S,V,U,B,D,I,Z, and C. The U bit is unused by the 6502 instruction set.

Module DISASM is used to disassemble code stored in a given range of memory. The code is disassembled in a pure form without reference to any source lines or any labels used in the source lines. Any block of code can be disassembled as long as valid opcodes are encountered in the first byte of each instruction. DISASM calls module DINSTR to disassemble a single instruction. DINSTR uses arrays OPCD, MD, and LEN to break an opcode into its opcode mnemonic, mode, and length. The TRACE module also calls DINSTR to disassemble code not corresponding to any source lines (for printing purposes only).

Other modules used by the simulator, but not mentioned so far, are IAND, IOR, IEOR, HVAL, BVAL, CH2, CH4, and CH8. Functions IAND, IOR, and IEOR perform the bit by bit logic operations of "and", "or", and "exclusive or", respectively, on a pair of byte operands. Functions HVAL and BVAL are used to convert hexadecimal values and binary values, respectively, into integer values. These are the same functions called by the assembler to perform these tasks. The simulator uses functions CH2, CH4, and CH8 to convert integer values to hexadecimal character strings of 2, 4, and 8 characters, respectively. CH2 and CH4 were also used by the assembler.

Some commands are executed directly by the simulator without the need of calling other modules. Prominent among these are the Z and O commands. The Z command is used to reset the simulator - all registers and counters

are zeroed out, except the SP register which is set to FF (indicates an empty stack). The O command is used to turn the register printing switch PRTSRC on or off. PRTSRC is a logical variable and is considered "on" when PRTSRC stores value .TRUE. and "off" otherwise. When the switch is on, the values of all registers are automatically displayed along with the source lines during program tracing via the T or W commands.

5.4 I/O PORTS AND INTERRUPTS

The simulator uses 5 special addresses for I/O ports and data direction registers:

9110 - PORT B

9111 - PORT A (with handshaking)

9112 - DDRB (Data Direction Register B)

9113 - DDRA (Data Direction Register A)

911F - PORT A (without handshaking)

The above memory addresses are hexadecimal values. Logical variable SHAKE is used to keep track of which PORT A is currently active. The intention here is to simulate on a limited scale the 6522 Versatile Interface Adapter I/O chip so that the user can perform I/O operations and quickly test port values via the debugger commands (the VP command primarily). For those instructions requiring input from memory, module TRACE must determine whether the effective address refers to a port before the instruction

is executed. If so, a value must be input from the terminal. After the instruction is executed, for those instructions that change memory, the effective address is again checked against port addresses. If a value has been output to a port, the value is printed at the terminal. The data direction registers are used to determine which bits in the ports can be changed on an I/O operation.

The simulator permits interrupts via the CONTROL Y character during source code execution.. This feature works only on VAX computers since several VAX system routines are called by module TRACE to force the interrupt while instructions are being executed. Similarly, the simulator permits the use of the CONTROL C character to stop execution of source code. Control returns to EXECUT, the top level module in the simulator). Modules INITT, CDECL, CINT, YDECL, and YINT are used to implement these VAX features. On non-VAX computers, calls to these modules would have to be omitted.

REFERENCES

Esformes, M. "Control," Micro: The 6502/6809 Journal, June, 1984, pp. 36-42.

Freiberger, P. "Emulate and Cross Assemble to Transport Software," Infoworld, October 19, 1981, p.21

Little, T. "The Visible Computer: 6502," Peelings II, Vol. 4, No. 5 (1983), pp. 60-61.

Lloyd, D. "6502 Cross Assembler for Z80 CP/M," Online Today, December, 1984, p. 39.

Mashey, J.R., and G.M. Campbell. ASSIST Introductory Assembler User's Manual, Computer Science Department, Pennsylvania State University, March 1974.

Nagata, W.M., and D.S. Miller, "An Interactive Simulator for the KIM-1 Microprocessor," Simulation, January, 1981, pp. 21-33.

Ruppert, J. "6502 Tracer," Elektor, February, 1984, pp. 22-23.

Wittenberg, R. "Portable Emulator Priced Under $600," Electronic Engineering Times, August 27, 1984, p.61

Zeck, S.R., "Cross Assembler for Microcomputer Programming" (unpublished report, Department of Computer Science, Washington State University, January, 1979).

APPENDIX A

6502 ASSIST USER'S GUIDE

6502 ASSIST is an interactive program designed to help you write and debug 6502 source code. The ASSIST package consists of three major components: an editor, an assembler, and a simulator. The editor is used to create a 6502 source file or to modify an existing 6502 source file.

The editor has special features that permit you to check individual source lines for 6502 syntax errors or the entire file for any assembler errors. The assembler is used to create a 6502 object file from your source file. The assembler also prints an assembly listing. The simulator is used to run-time test your source code and its corresponding object code. The simulator has extensive debugging facilities to help you locate and correct run-time errors.

1. USING 6502 ASSIST

Before you can use the 6502 ASSIST program you must have an account on the VAX 11/780 computer system at Montana Tech. To run the 6502 ASSIST program, you must first log in on the VAX and type the command

```
$ RUN [VWDANIEL]6502
```

where the '$' character is typed by the VAX operating system and the rest by you. At this point the 6502 ASSIST

prompt '<' should appear on your terminal. You can now enter any of the 6502 ASSIST commands shown in Table I. For example, if you wished to create a new file called SAMPLE.SRC in your directory, you could issue the following command:

<ED SAMPLE

where **boldface** type indicates characters printed by the computer. Note that a file extension of .SRC is assumed by the ASSIST program. The ED command calls the 6502 editor to create or modify a file.

2. USING THE EDITOR

The editor views a file as a collection of lines. Each line has a line number to faciltiate access to that line. Normally lines are numbered in multiples of 100, although any line numbers from 00001 to 99999 are permitted.

The editor is entered from 6502 ASSIST by the ED command as previously mentioned. The editor will prompt you for the file name if not given with the ED command. The editor will also prompt you for the file status (OLD or NEW) and whether you wish to turn the syntax checking switch on (YES or NO). When the switch is turned on, new lines being inserted into the file will be automatically checked for proper 6502 syntax. When the switch is off, the editor functions much like any general purpose

line editor. The syntax checking mechanism can be switched on or off at any time by the O command. If the file is a NEW file, the editor automatically goes into insertion mode (line number 00100 appears on your terminal and the editor waits for you to enter the first line). If the file is an OLD file, the editor prompt '*' appears on your terminal and the editor waits for you to enter an editor command. A complete list of all editor commands appears in Table 2.

A sample editor session is shown on the next page. SAMPLE.SRC is the name of the source file. **Boldface** type indicates characters printed by the computer. The program shown computes the sum of the first 5 integers and stores the sum in location 0021.

```
<ED SAMPLE
OLD OR NEW? N
SYNTAX CHECK EACH NEW LINE? (YES OR NO)
00100               ORG $900
00200               CLC
00300               LDX #5
00400     LOOP      STX $
00500               ADD $20
    ***   ADD    IS AN INVALID OPCODE
          PLEASE RE-ENTER SOURCE LINE
00500               ADC $20
00600               DEX
00700               BNE LOOP
00800               STA $21
00900               BRK
01000      ^Z
*
```

Note the use of the Control Z character to escape the insertion mode. In line 500 the editor detected a syntax error and requested that the line be retyped. If the

syntax checking mechanism had not been switched on, the error would have been left undetected until an explicit syntax check of the file was performed (via the C, K or A editor commands). Of course the assembler would catch all syntax errors at assembly time.

Most editor commands permit line number ranges. Thus to obtain a listing of the source file, type the command

```
*P100,900
```

or better yet type

```
*p^,*
```

where '^' denotes the first line and '*' denotes the last line (the current line is denoted by '.'). These special symbols are permitted anywhere line number parameters are valid. The sample editor session continues as shown.

```
*p^,*
00100                   ORG $900
00200                   CLC
00300                   LDX #5
00400       LOOP        STX $20
00500                   ADC $20
00600                   DEX
00700                   BNE LOOP
00800                   STA $21
00900                   BRK
*C

***    NO ASSEMBLY ERRORS    ***

*W
*
```

Note the use of the C command (without any parameters) to check the source file for assembly errors. This

command also creates the object file SAMPLE.OBJ, so a formal assembly is not needed unless you want an assembly listing. The W command writes a new version of the source file SAMPLE.SRC into your directory. Control remains in the editor, so the editing session may continue. Use the E command to save the source file and exit the editor.

Most of the editor commands in Table 2 are used in a straight forward manner. As an example, let's change all register X references to register Y references in the sample program. This could be accomplished by repeatedly using the find and substitute commands as shown below.

```
*FX\^
00300                   LDX #5
*SX\Y\
00300                   LDY #5
*F
00400      LOOP         STX $20
*S
00400      LOOP         STY $20
*F
00600                   DEX
*S
00600                   DEY
*F
STRING NOT FOUND
*P^,*
00100                   ORG $900
00200                   CLC
00300                   LDY #5
00400      LOOP         STY $20
00500                   ADC $20
00600                   DEY
00700                   BNE LOOP
00800                   STA $21
00900                   BRK
*
```

Note the use of the find and substitute commands

without any parameters. The find/substitute commands "remember" their last search strings. So after an initial use with parameters, they can be subsequently invoked without parameters to repeat the previous command at the current location. Most editor commands can be invoked without line number parameters. The default line is the current line. As an example, suppose we wished to delete line 800. The safest way to do this would be to type:

```
*P800
00800          STA $21
*D
1 LINE(S) DELETED
*
```

An alternate way would be to type:

```
*D800
1 LINE(S) DELETED
*
```

To insert line 800 back into the file, now type:

```
*I800
00800          STA $21
*
```

Finally, let's change all the Y's back to X's with one command:

```
*SY\X\^,*
00300          LDX #5
00400    LOOP  STX $20
00600          DEX
*
```

It is recommended that you use the find command to

locate a search string before attempting wholesale substitutions. Careless use of the substitute, delete, and replace commands can lead to mutilation of the source file.

The editor supports cut and paste type operations via the move and copy commands. Whole ranges of lines can be moved or copied to new locations. The editor automatically renumbers the inserted lines based on even spacing in the target range. As an example, let's move lines 500-800 to a new range between lines 300 and 400. The editor session continues as shown.

```
*M500,800,300
  4 LINE(S) MOVED TO RANGE 00320:00380
*P^,*
00100                  ORG $900
00200                  CLC
00300                  LDX #5
00320                  ADC $20
00340                  DEX
00360                  BNE LOOP
00380                  STA $21
00400    LOOP          STX $20
00900                  BRK
*Q
EXITING EDITOR - NO CHANGES
<
```

Note the use of the Q command to exit the editor without writing a new version of file SAMPLE.SRC in your directory. The Q command should be used anytime you have made major mistakes in editing your source file and it would be easier to start all over again (with the previous version).

3. USING THE ASSEMBLER

The assembler is entered from 6502 ASSIST by typing the A command. The assembler will prompt you for the source file name if not given with the A command. The assembler also can be called by the editor via the A command. In this case no file name is specified since it is assumed that you want to assemble the source file that you are currently editing.

The assembler creates the object file and prints an assembly listing. Suppose we want to assemble the source file SAMPLE.SRC created by the previous editor session. The assembler could be called from 6502 ASSIST by typing the command:

```
<A SAMPLE
```

The assembler would then proceed to create the object file SAMPLE.OBJ in your directory and print an assembly listing on your terminal as shown.

```
                    *** ASSEMBLY LISTING ***

 LC          CODE           STMT              SOURCE LINE

0000:                         1                     ORG $900
0900:    18                   2                     CLC
0901:    A2 05                3                     LDX $5
0903:    86 20                4    LOOP             STX $20
0905:    65 20                5                     ADC $20
0907:    CA                   6                     DEX
0908:    D0 F9                7                     BNE LOOP
090A:    85 21                8                     STA $21
090C:    00                   9                     BRK

          *** SUCCESSFUL ASSEMBLY - NO ERRORS ***

EXITING ASSEMBLER
<
```

If any errors are detected, they would be noted at the appropriate place in the listing. The assembly listing always ends with a summary message indicating the number of errors detected. If there are no errors, then the source program is ready for run-time testing via the 6502 simulator. If any errors are detected, it will be necessary to re-enter the editor, correct the errors, and re-assemble the source file. The check (C) command of the editor is particularly useful for a quick assembly of the source file without having to wait for a full assembly listing or leave the editor. A further advantage of the C command is that any errors detected are listed using the **editor** line numbers.

The assembler recognizes all standard 6502 pseudo-operations. In addition, ORG may be used for the origin pseudo-op (*=) and EQU may be used for the equate pseudo-op (=).

Labels must start with a letter and contain only letters and digits. The assembler truncates all labels to the first six characters. So you probably shouldn't use labels longer than six characters. The label, operation code, and address field must be separated by blanks or tab characters, or both. Embedded blanks are not permitted within the address field.

Arithmetic expressions are permitted within the address field. The only valid expressions are those containing numbers and labels separated by +, -, *, and /

arithmetic operators. Numbers may be expressed in binary (% prefix), octal (@ prefix), hexadecimal ($ prefix), or decimal (no prefix). Addresses may be given as an offset from the program counter (* symbol). A single ASCII character preceded by an apostrophe is also allowed as an address. Use of the END pseudo-op is entirely optional.

4. USING THE SIMULATOR

For the simulator to function correctly, you must have a copy of the source file and its matching object file in your directory. **If any changes are made to the source file, it is absolutely imperative that you re-assemble the source file to obtain an up-to-date object file in your directory before attempting to use the simulator.** Failure to do this will surely lead to confusion since the source statements printed by the simulator will not correspond to the object code being executed. The source file can be assembled by the A,C, or K commands of the editor or the A command of 6502 ASSIST.

The simulator is entered from 6502 ASSIST by the X command. The simulator will prompt you for the source file name if not given with the X command. The source file and its corresponding object file are then loaded into the simulator. The object code is actually loaded in the simulator's 64K RAM "memory". The simulator can also be entered from the editor via the X command. In this case no file name is specified since it is assumed you want to

run-time test the source file that you are currently editing (don't forget to assemble the source file first).

Suppose we want to run-time test the source file SAMPLE.SRC and its corresponding object file SAMPLE.OBJ. The simulator could be called from 6502 ASSIST by typing the command:

```
<X SAMPLE
9 SOURCE LINES LOADED
.
```

After the source file and its matching object file have been loaded, the simulator prompt '.' will appear on your terminal. At this point you can enter any of the simulator commands listed in Table 3. Let's print the source file to make sure the file was loaded properly.

```
.P^,*
  1                ORG $900
  2                CLC
  3                LDX #5
  4    LOOP        STX $20
  5                ADC $20
  6                DEX
  7                BNE LOOP
  8                STA $21
  9                BRK
.
```

Note that the simulator, like the assembler, renumbers the lines in the source file beginning with line number 1. We can check that the object code was loaded correctly into memory by typing:

```
.M900
0900: 18 A2 05 86 20 65 20 CA D0 F9 85 21 00 00 00 00
```

or by typing:

```
.MW900,90F
0900:  18
0901:  A2
0902:  05
0903:  86
0904:  20
0905:  65
0906:  20
0907:  CA
0908:  D0
0909:  F9
090A:  85
090B:  21
090C:  00
090D:  00
090E:  00
090F:  00
.
```

Note that the memory (M) command normally prints 16 memory locations per line while the memory word (MW) command prints only one location per line. Both commands require hexadecimal parameters. In fact, all memory location parameters must be given in hexadecimal form, while all line number parameters must be given in decimal form. The special symbols '^','.', and '*' may be used as line number parameters to denote the first line, the current line, and the last line, respectively. Similarly, these special symbols may be used as memory location parameters. However in this case, '^' always refers to memory location 0000 and '*' always refers to memory location FFFF.

Now let's execute our source program. If there are no errors, the program should add the first 5 integers and store the result in memory location 0021. There are five different execution modes available on the simulator. We will begin by using the GO mode (G command). The GO mode executes the source program at full speed, stopping only at program breaks or invalid opcodes. The simulator session continues as shown.

```
.G900
*** PROGRAM BREAK AT LINE      9

.MW21
0021:   0F

.R

 PC   A   X   Y   SP SVUBDIZC
090C 0F  00  00  FF 00000010

.
```

Note that memory location 0021 contains a value of 0F hexadecimal or 15 decimal. This is the correct sum. The register (R) command displays the current values of all the registers. The program status register P is displayed as 8 bits, where the U bit is unused by the 6502 microprocessor.

The execution count for each statement can now be obtained by the K command:

```
.K^,*
  1(000000)                ORG $900
  2(000001)                CLC
  3(000001)                LDX #5
  4(000005)      LOOP      STX $20
  5(000005)                ADC $20
  6(000005)                DEX
  7(000005)                BNE LOOP
  8(000001)                STA $21
  9(000000)                BRK
.
```

Note that the ORG statement is not executed since it generates no object code in memory (ORG only sets the program counter). Also note that the BRK statement is not executed since the GO mode halts at all program breaks. The BRK instruction can be executed at this point by typing a G command without any parameters or just by typing a carriage return (which continues the previous command at the current location). However, doing this is not advised unless you have placed an appropriate interrupt routine in memory and the starting address of this routine in memory locations FFFE and FFFF.

The N command gives the total number of statements executed as well as the statement limit:

```
.N
STATEMENT LIMIT:          10000
NUMBER EXECUTED:             23
```

The statement limit is the maximum number of statements that the simulator will execute (the default limit is 10000). The statement limit can be changed at any time by the N command. For example, to set the

statement limit at 500 type:

```
.N500
STATEMENT LIMIT:          500
NUMBER EXECUTED:           23
```

When a program does not terminate as expected, it would be highly desirable to know where in the source code (or where in memory) control was when the program died. The DUMP command provides this information in the form of two listings:

```
.DUMP

*** LAST 10 SOURCE INSTRUCTIONS EXECUTED ***

 7              BNE LOOP
 4     LOOP     STX $20
 5              ADC $20
 6              DEX
 7              BNE LOOP
 4     LOOP     STX $20
 5              ADC $20
 6              DEX
 7              BNE LOOP
 8              STA $21

*** LAST 10 BRANCH INSTRUCTIONS EXECUTED ***

 7              BNE LOOP
 7              BNE LOOP
 7              BNE LOOP
 7              BNE LOOP
 7              BNE LOOP
```

In our sample program, the branch instruction on line 7 was executed only 5 times and there were no other branch instructions in the program. Note that the BRK instruc-

tion on line 9 was not executed, as was expected.

The GO mode provides the least control over the execution process. For more control use the quick trace mode (Q command) or the trace mode (T command). For maximum control use the walk mode (W command). The jump mode (J command) is used to jump through a subroutine call.

The two tracing modes stop execution at program breaks, software breakpoints, non-source instructions, and invalid opcodes. Program execution can be continued by just typing a carriage return. Program breakpoints are the BRK instructions in your source code. Software breakpoints are artificial breakpoints maintained by the simulator and recognized only during program execution by the two tracing modes. Software breakpoints can be inserted at any source line by the break (B) command.

For example, to insert software breakpoints at lines 4 and 8 of our sample program type:

```
.B4

.B8
```

To verify all breakpoints currently set type:

```
.VB
STMT      BREAK AFTER

  4            0
  8            0
```

It is possible to set software breakpoints that only

"break after" a specified number of executions of the given source statement. For example, if we wished to set a breakpoint at line 6 that breaks only after three executions of line 6 (ie, breaks just before the 4th execution) we would type:

```
.B6,3
```

However, let's not do this. So to remove the above breakpoint type:

```
.RB6
1 SOFTWARE BREAKPOINT(S) REMOVED
```

We are back to our original breakpoints at lines 4 and 8. These breakpoints break after 0 executions, so control stops just before either line is about to be executed.

Now let's trace program execution using the quick trace mode. First reset the simulator via the Z command. This command zeroes out all registers and counters, except the stack pointer SP which is set to FF (an empty stack). The simulator session continues as shown.

```
.Z

.Q900
*** SOFTWARE BREAK AT LINE    4

.R

 PC   A  X  Y  SP  SVUBDIZC
0903 00 05 00 FF  00000000

.Q
*** SOFTWARE BREAK AT LINE    4
```

```
.R

 PC   A  X  Y  SP   SVUBCIZC
0903 05 04 00 FF   00000000

.G
***  PROGRAM BREAK AT LINE    9

.R

 PC   A  X  Y  SP   SVUBDIZC
090C 0F 00 00 FF   00000010
```

Note the use of the R command to print out the register values at each software break. Since execution seems to be proceeding as expected after two software breaks, we switch to the GO mode to skip the remaining software breaks. The final value of register A contains the correct sum.

Now that we have seen how the quick trace mode operates, we will try the (slow) trace mode. First we remove all the software breakpoints (via the RB command) since they won't be needed and we reset the simulator. We will make the trace mode even slower by turning the register printing switch on (via the O command). Initially this switch is turned off, but its setting can be reversed at anytime by the O command. The register switch affects the way source lines are displayed when using the trace, walk, or jump execution modes. When switched on, the values of all the registers are displayed along with the source line about to be executed. When switched off, only the source line is displayed. Note

that the source line is executed after it is displayed. The register values are not automatically printed for the other two execution modes regardless of the switch setting. The simulator session continues as follows:

```
.RB^,*
  2 SOFTWARE BREAKPOINT(S) REMOVED

.Z

.O
REGISTER PRINT SWITCH ON

.T900

 PC   A   X   Y   SP SVUBDIZC   STMT      SOURCE LINE

0900  00  00  00  FF 00000000     2            CLC
0901  00  00  00  FF 00000000     3            LDX  #5
0903  00  05  00  FF 00000000     4    LOOP    STX $20
0905  00  05  00  FF 00000000     5            ADC  $20
0907  05  05  00  FF 00000000     6            DEX
0908  05  04  00  FF 00000000     7            BNE  LOOP
0903  05  04  00  FF 00000000     4    LOOP    STX  $20
0905  05  04  00  FF 00000000     5            ADC  $20
0907  09  04  00  FF 00000000     6            DEX
0908  09  03  00  FF 00000000     7            BNE  LOOP
0903  09  03  00  FF 00000000     4    LOOP    STX  $20
0905  09  03  00  FF 00000000     5            ADC  $20
0907  0C  03  00  FF 00000000     6            DEX
0908  0C  02  00  FF 00000000     7            BNE  LOOP
0903  0C  02  00  FF 00000000     4    LOOP    STX  $20
0905  0C  02  00  FF 00000000     5            ADC  $20
0907  0E  02  00  FF 00000000     6            DEX
0908  0E  01  00  FF 00000000     7            BNE  LOOP
0903  0E  01  00  FF 00000000     4    LOOP    STX  $20
0905  0E  01  00  FF 00000000     5            ADC  $20
0907  0F  01  00  FF 00000000     6            DEX
0908  0F  00  00  FF 00000010     7            BNE  LOOP
090A  0F  00  00  FF 00000010     8            STA $21
090C  0F  00  00  FF 00000010     9            BRK
***   PROGRAM BREAK AT LINE       9
```

As you can see, the trace mode with the register print switch on provides for a very detailed trace of program execution. It is also quite time consuming for all but

the simplest of programs. A good procedure would be to use the trace mode on only a part of your source code. This can easily be done by setting software breakpoints at the beginning and at the end of the section you wish to trace. Then begin tracing via the quick trace mode, switch to the trace mode at the first breakpoint, and switch back to the quick trace or GO mode at the second breakpoint. The execution modes may be used in any order to trace through a complete program.

An additional feature of the two tracing modes (Q and T commands) is that they will stop at all non-source instructions. This feature is useful for run-time testing of programs that "run wild" through memory. Program tracing can be continued by just typing a carriage return.

The walk mode executes one instruction at a time under your control. The walk mode is initiated by the W command, but may be continued by repeatedly typing the carriage return key. First we reset the simulator (via the Z command) and turn the register printing switch off (via the O command). The simulator session continues as shown.

```
.Z

.O
REGISTER PRINT SWITCH OFF

.W900

STMT          SOURCE LINE

  2                CLC

.
  3                LDX #5

.
  4   LOOP         STX $20

.
  5                ADC $20

.
  6                DEX

.
  7                BNE LOOP

.G
 *** PROGRAM BREAK AT LINE   9
```

Although the carriage returns aren't explicitly shown in this example, they were typed after each prompt (the period). Note that we switched to the GO mode to finish program execution at line 7. The line displayed at each step is the next instruction to be executed. Typing a carriage return will execute the instruction and display the next line in execution order.

The final execution mode is the jump mode. This mode is used to execute all instructions within a subroutine call. The jump mode can be invoked at any time via the J command (with no parameters). If the next instruction to be executed is a JSR instruction, then the entire

subroutine call is executed. If the next instruction is already within a subroutine call, but is not a JSR instruction, then the remainder of the current subroutine call is executed. Any subroutine calls nested within the current call are also executed. If the next instruction is not a JSR instruction and is not within a subroutine call, then the jump mode behaves like the walk mode, executing the current instruction and listing the next one. The jump command will not be illustrated here since our sample program contains no subroutine calls.

The simulator provides a way to stop program execution while in progress. This is done by typing a Control C character. The simulator prompt "." will appear and any simulator command may be given at this point. Typing a carriage return will continue program execution.

The simulator also provides a way to signal interrupt requests during program execution. This feature is useful for testing interrupt routines as well as source program handling of interrupt requests. Typing a Control Y character during source code execution signals an interrupt request. Multiple interrupt requests are permitted. The trace mode is probably the most practical execution mode to use for initial testing of interrupt requests since it is fairly slow and prints each instruction. To give an illustrated example would go beyond the scope of this introductory guide.

Suppose we want to augment our source code to compute the sum of the first 10 integers and to store the result in location 0022. We could, of course, exit the simulator, re-enter the editor, make the necessary changes, exit the editor, re-assemble the source file, and re-enter the simulator to run the augmented file. We will not do this, however, since we wish to demonstrate the power of the change (C) and fill (F) commands to accomplish the same thing without leaving the simulator.

First we reset the simulator. Next we fill location 090B with value 22 hexadecimal. This changes the source code in memory corresponding to line 8, but not line 8 itself. Then we run the augmented code in walk mode. After line 3 has been executed (line 4 will be displayed), we change the value stored in register X to 0A hexadecimal (10 decimal) and continue execution via the quick trace mode. The simulator session continues as shown.

```
.Z

.F22,90B
   1 LOCATION(S) FILLED

.W900

STMT            SOURCE LINE

  2                   CLC

.
  3                   LDX #5

.
  4     LOOP          STX $20

.CX   0A

.Q
** NONSOURCE INSTRUCTION ENCOUNTERED AT LOCATION 090A

.Q
** PROGRAM BREAK AT LINE    9

.MW22
0022:  37
```

Note that execution via the quick trace mode halted at the instruction stored in memory location 090A. Recall that we changed the second byte (location 090B) of this instruction, so it no longer is a source instruction. Execution can be continued, however, by any of the execution modes as long as the instruction has a valid opcode. Note that the sum stored in memory location 0022 is 37 hexadecimal (55 decimal). This is the correct sum of the first 10 integers. The simulator will consider the instruction stored in memory location 090A a non-source instruction for the remainder of this simulator session, even if we change the value in location 090B back to its

original value.

Next we give a demo of the hunt (H) command. Suppose we wish to hunt for all bytes in memory with value 20 hexadecimal. First we must position the memory pointer to the first byte of memory. This can be done by typing:

```
.MW^
0000:  00

.H20
```

or by typing:

```
.H20,^
```

The following simulator session uses the second method.

```
.H20,^
0904:  20

.H
0906:  20

.H
STRING NOT FOUND
```

Note that once the search is initiated, the search may be continued by just typing the letter H without any parameters since the hunt command remembers its last search string.

The simulator has two commands for matching source statements to memory locations. The address (A) command shows where in memory the code for a given range of source lines is stored. The source (S) command lists the

source statements corresponding to a given range of memory. Using these commands we can find the current status of our files in memory.

```
.A^,*
STMT      1   NO LOC
STMT      2   AT LOC 0900
STMT      3   AT LOC 0901
STMT      4   AT LOC 0903
STMT      5   AT LOC 0905
STMT      6   AT LOC 0907
STMT      7   AT LOC 0908
STMT      8   NO LOC
STMT      9   AT LOC 090C

.S900,910
0900:  2               CLC
0901:  3               LDX #5
0903:  4  LOOP         STX $20
0905:  5               ADC $20
0907:  6               DEX
0908:  7               BNE LOOP
090C:  9               BRK
```

Note that source line 8 is considered to have no code stored in memory since we changed one byte of the instruction (location 090B) earlier. Thus locations 090A-090B no longer correspond to any source code.

Sometimes it is helpful to disassemble code in a section of memory without regard to the source lines. This feature is particularly helpful when you have changed the source code stored in memory, but not the source lines (only the editor can change the source lines). Use the D command to get a listing of the instructions currently stored in memory. A disassembly of memory locations 900 to 90F is shown below.

```
.D900,90F
0900:   CLC
0901:   LDX #$05
0903:   STX $20
0905:   ADC $20
0907:   DEX
0908:   BNE $0903
090A:   STA $22
090C:   BRK
090D:   BRK
090E:   BRK
090F:   BRK
.
```

The simulator can handle multiple source files. The simulator always prompts you, if necessary, for the first file name upon entering the simulator. Additional source files may be loaded using the L command. **You must have a matching, up-to-date object file in your directory for each source file you load.** The simulator will automatically load the corresponding object files for you. The simulator also checks memory for overwriting of source code. If the code for any source line overwrites the code for a previously loaded source line, a warning message is printed and the new source code is loaded anyway. This is a useful feature when multiple files must be loaded before run-time testing can begin (as is typical with modular programming).

The carriage return and backslash keys are generally used to repeat the previous command in a forward direction or in a backward direction. For forward motion, repeatedly press the carriage return key. For backward motion, repeatedly press the backslash and carriage return

keys in succession. There are some exceptions. For those commands for which only forward motion makes sense (for example, the execution commands), the backslash and carriage return keys operate only in a forward direction. Following some commands, primarily the break, hunt, fill, load, and dump commands, the carriage return/backslash keys do not repeat the previous command, but only print source lines or memory locations in a forward or backward direction. With a little practice you will no doubt become familiar with the proper use of these keys.

Next we take a brief look at the I/O facilities of the simulator. The simulator reserves five memory addresses for I/O operations:

9110	PORT B	
9111	PORT A	(with handshaking)
9112	DDRB	(data direction register B)
9113	DDRA	(data direction register A)
911F	PORT A	(without handshaking)

The handshaking itself is not simulated. The intention here is to simulate on a limited scale the 6522 Versatile Interface Adapter (VIA) chip. Ports A and B can be used as either input or output ports. In fact, each pin (bit) of port A or port B can be individually selected as an input pin or an output pin. The data direction registers DDRA and DDRB are used to accomplish this. A zero bit in the data direction register makes the

corresponding pin an input pin, while a one bit makes the corresponding pin an output pin. For example, to make port A an input port (all pins in input mode) set DDRA to 00000000 binary or 00 hexadecimal. To make port B an output port (all pins in output mode) set DDRB to 11111111 binary or FF hexadecimal. As an illustration, consider the following program which reads an input value from port A and outputs the same value to port B.

```
 1      PORTA      EQU $9111
 2      PORTB      EQU $9110
 3      DDRA       EQU $9113
 4      DDRB       EQU $9112
 5                 ORG $900
 6                 LDA #%00000000
 7                 STA DDRA
 8                 LDA #%11111111
 9                 STA DDRB
10                 LDA PORTA
11                 STA PORTB
12                 BRK
```

After assembling the source file, the simulator can be used to run the program in any of the execution modes. The GO mode is used in the following simulator session.

```
.G900

INPUT  PORT A:  00110110

OUTPUT PORT B:  00110110

*** PROGRAM BREAK AT LINE   12

.VP

VALUE PORT A:   00110110       VALUE PORT B: 00110110
CURRENT DDRA:   00000000       CURRENT DDRB: 11111111
```

The VP command is used to verify the current port values and data direction register values.

Although we will not illustrate them here, there are two commands for listing values stored in memory as ASCII characters. These are the inspect (I) command and the inspect word (IW) command. The first normally prints 16 memory locations per line, the second only one memory location per line. These commands are useful for displaying text stored in memory.

Finally, there is a command to exit the simulator. Just type the letter E. Control returns to 6502 ASSIST, the top level program. Type a second E, you exit 6502 ASSIST and control returns to the VAX operating system (VMS). The simulator does not create any files. Neither does it alter any files created by the editor or the assembler.

5. CONCLUSION

Having read each of the preceding sections, you should be ready to enter your own 6502 source program, check it for syntax errors, and run-time test it for correct performance. Of course, a thorough knowledge of the 6502 instruction set would be essential. There are several good books available on 6502 assembly language programming and it is recommended that you have at least one handy for a reference. Perhaps you would like to try out some of the sample programs contained in these books on the 6502

simulator.

Tables 1, 2, and 3 that follow should prove sufficient for most purposes in using the 6502 ASSIST package. If not, refer to the appropriate sections of this guide for more detailed information.

One final note: you do not have to use the editor supplied with the 6502 ASSIST package. Any editor will suffice. You will have to give up the syntax checking features of the 6502 editor, however, if you use another editor. Once you have created the source program, then enter 6502 ASSIST to assemble and simulate your source code.

Table 1

Summary of 6502 ASSIST Commands

COMMANDS	FORMS	DESCRIPTION
ED (EDIT)	ED	Edits a file (editor prompts for file name)
	ED TEST	Edits file TEST.SRC
	ED T.OLD	Edits file T.OLD
A (Assemble)	A	Assembles a 6502 source file (assembler prompts for file name)
	A TEST	Assembles source file TEST.SRC
	A T.OLD	Assembles source file T.OLD
X (eXecute)	X	Executes a 6502 source file (simulator prompts for file name)
	X TEST	Executes source file TEST.SRC
	X T.OLD	Executes source file T.OLD
E (Exit)	E	Exits 6502 ASSIST program

6502 ASSIST NOTES

1. To use the 6502 ASSIST program, first log in on any VAX terminal and type the command $ RUN [VWDANIEL]6502. The 6502 prompt '<' should now appear on your terminal. At this point you can enter any of the 6502 commands listed above.

2. To use the 6502 EDITOR, type the ED command in response to the 6502 ASSIST prompt. The editor will prompt you for the file name if not already given, the file status (OLD or NEW), and whether you wish to turn the syntax checking switch on (YES or NO). The 6502 syntax checking mechanism, when switched on, only checks new lines being inserted by the I or R editor commands. If the file is a NEW file, the editor automatically goes into insertion mode (line numbers appear on your terminal). If the file is an OLD file, the editor prompt '*' appears on your terminal. At this point you can enter any of the 6502 EDITOR

commands (see appropriate table). The 6502 EDITOR is used to create 6502 source files.

3. To use the 6502 ASSEMBLER, type the A command in response to the 6502 ASSIST prompt. The assembler will prompt you for the file name if not already given. At this point the assembler will print an assembly listing of your source program together with any assembly errors found. The 6502 ASSEMBLER can also be called directly from the 6502 EDITOR in the same manner, except that no file name is specified. The 6502 ASSEMBLER is used to create 6502 object files.
4. To use the 6502 SIMULATOR, type the X command in response to the 6502 ASSIST prompt. The simulator will prompt you for the file name if not already given. The simulator prompt '.' should now appear on your terminal. At this point you can enter any of the 6502 SIMULATOR commands (see appropriate table). The 6502 SIMULATOR can also be called directly from the 6502 EDITOR in the same manner, except that no file name is specified. The 6502 SIMULATOR is used to execute and debug 6502 source programs. For the simulator to function correctly, the corresponding object files must also exist in your directory. Therefore, you should always assemble a source file before attempting to execute it (this may be conveniently done with the C command of the editor if an assembly listing is not needed).

Table 2

Summary of Editor Commands

COMMAND	FORMS	DESCRIPTION
P (Print)	P	Prints current line
	P500	Prints line 500
	P500,800	Prints lines 500 to 800
I (Insert)	I	Inserts new line just after current line
	I500	Inserts new line at line 500 if there is no line 500, otherwise inserts just after line 500
	I500,10	Inserts new lines beginning at line 500 in increments of 10
D (Delete)	D	Deletes current line
	D500	Deletes line 500
	D500,800	Deletes lines 500 to 800
R (Replace)	R	Replaces current line
	R500	Replaces line 500
	R500,800	Replaces 500 to 800
N (reNumber)	N	Renumbers all lines in increments of 100
	N10	Renumbers all lines in increments of 10
E (Exit)	E	Exits editor and writes edited file to disk
Q (Quit)	Q	Exits editor but does not write edited file to disk
W (Write)	W	Writes edited file to disk but does not exit editor
Z (append)	Z	Appends an existing file to current file (editor prompts for file name)
	Z P2.FOR	Appends file P2.FOR to current file
S (Substitute)	SCAT\FISH\	Substitutes first occurrence of string 'CAT' with string 'FISH' (search begins at current line)
	SCAT\FISH\500	Substitutes first occurrence of string 'CAT' with string 'FISH' (search begins at line 500)
	SCAT\FISH\500,800	Substitutes first occurrence of string 'CAT' with string 'FISH' in all lines in the given range (lines 500 to 800)

Table 2 (continued)

COMMAND	FORMS	DESCRIPTION
T (substitute)	TCAT\FISH\	Substitutes all occurrences of string 'CAT' with string 'FISH' in the first line found containing string 'CAT' (search begins at current line)
U (substitute)	UCAT\FISH\ UCAT\FISH\500 UCAT\FISH\500, 800	Same as S command, except no lines are printed
V (substitute)	VCAT\FISH\ VCAT\FISH\500 VCAT\FISH\500, 800	Same as T command, except no lines are printed
F (Find)	FCAT\	Finds first occurrence of string 'CAT' (search begins at line following current line)
	FCAT\500	Finds first occurrence of string 'CAT' (search begins at line 500)
	FCAT\500,800	Finds all occurrences of string 'CAT' in lines 500 to 800
\ (backslash)	\	Prints previous line
cr (Carriage Return)	cr	Prints next line
M (Move)	M500,800,300	Moves lines 500 to 800 to a new location beginning at line 300 or just after line 300 (lines 500 to 800 are deleted)
Y (copY)	Y500,800,300	Copies lines 500 to 800 to a new location beginning at line 300 or just after line 300 (lines 500 to 800 are not deleted)
C (Check)	C	Checks all lines in file for assembly errors (assembles source file and creates object file)
	C500	Checks line 500 for syntax errors

Table 2 (continued)

COMMAND	FORMS	DESCRIPTION
C (Check) (continued)	C500,800	Checks lines 500 to 800 for syntax errors
K (checK)	K	Prints and checks all lines in file for assembly errors (assembles source file and creates object file)
	K500	Prints and checks line 500 for syntax errors
	K500,800	Prints and checks lines 500 to 800 for syntax errors
O (On/Off)	O	Turns syntax checking switch on/off
A (Assemble)	A	Assembles source file, creates object file, and prints an assembly listing
X (eXecute)	X	Executes object file created by A, C, or K commands

EDITOR NOTES

1. The editor permits special symbols to denote the first line (^), the current line (.), and the last line (*). Illustrated below are some examples using these special symbols:

I*	Inserts new lines at end of file
P^,*	Prints all lines in file
D.,*	Deletes all lines from current line to last line
R^,.	Replaces all lines from first line to current line
C.	Checks current line for syntax errors

2. To escape the insertion mode or replace mode, type a CONTROL Z.

3. To stop a listing, type a CONTROL O followed by a carriage return.

4. To recover from a CONTROL C (typed inadvertently), type CONT followed by a carriage return.

5. The find/substitute commands remember their last search strings.

Table 3

Summary of Simulator Commands

COMMAND	FORMS	DESCRIPTION
P (Print)	P	Prints current line
	P5	Prints line 5
	P5,8	Prints lines 5 to 8
M (Memory)	M	Prints 16 memory locations beginning at current location
	M1C40	Prints 16 memory locations beginning at location 1C40
	M1C40,1C90	Prints memory locations 1C40 to 1C90
MW (Memory Word)	MW	Prints current memory location
	MW1C40	Prints memory location 1C40
	MW1C40,1C90	Prints memory locations 1C40 to 1C90 (one per line)
I (Inspect memory)	I	Prints 16 memory locations in ASCII beginning at current location
	I1C40	Prints 16 locations in ASCII beginning at location 1C40
	I1C40,1C90	Prints memory locations 1C40 to 1C90 in ASCII
IW (Inspect Word)	IW	Prints current memory location in ASCII
	IW1C40	Prints memory location 1C40 in ASCII
	IW1C40,1C90	Prints memory locations 1C40 to 1C90 in ASCII (one per line)
G (Go)	G	Execute program beginning at current memory location
	G1C40	Execute program beginning at memory location 1C40
		Note: Go stops at programs breaks
Q (Quick trace)	Q	Execute program beginning at current memory location
	Q1C40	Execute program beginning at memory location 1C40
		Note: Quick trace stops at program breaks, software breakpoints, and nonsource instructions

Table 3 (continued)

COMMAND	FORMS	DESCRIPTION
T	T	Trace programs execution beginning at current memory location
(Trace)	T1C40	Trace program execution beginning at memory location 1C40 Note: Trace stops at program breaks, software breakpoints, and nonsource instructions
W	W	Executes current instruction and lists next instruction
(Walk)	W1C40	Lists instruction stored at memory location 1C40 (typing a carriage return will execute the instruction and list the next instruction)
J (Jump)	J	Executes all instructions in a subroutine call Note: Jump stops at program breaks
R (Register)	R	Displays the registers PC,A,X,Y, and SP in hexadecimal and P in binary
O (On/Off	O	Turns the register printing switch on/off (used in conjunction with the tracing commands T and W)
C (Change)	CPC 1C40 CA 00 CX 3F CY 05 CSP FF CP 80 CS 1 CV 0 (etc)	Changes PC register to 1C40 Changes A register to 00 Changes X register to 3F Changes Y register to 05 Changes SP register to FF Changes P register to 80 Changes S bit of P register to 1 Changes V bit of P register to 0 (etc)
F (Fill)	F9E	Fills current location with hex value 9E
	F9E,1C40	Fills location 1C40 with hex value 9E
	F9E,1C40,1C90	Fills location 1C40 to 1C90 with hex value 9E
H (Hunt)	H A2 BF	Hunts for first occurrence of hex string A2BF (search begins at memory location following current location)

Table 3 (continued)

COMMANDS	FORMS	DESCRIPTION
H (Hunt) (continued)	HA2BF,1C40	Hunts for first occurrence of hex string A2BF (search begins at memory location 1C40)
	HA2BF,1C40,1C90	Hunts for all occurrences hex string A2BF in memory locations 1C40 to 1C90
	H'GET'	Hunts for first occurrence of ASCII string GET (search begins at memory location following current location)
	H'GET',1C40	Hunts for first occurrence of ASCII string GET (search begins at memory location 1C40)
	H'GET',1C40,1C90	Hunts for all occurrences of ASCII string GET in memory locations 1C40 to 1C90
B (Break)	B	Sets software breakpoint at current line
	B5	Sets software breakpoint at line 5
	B5,100	Sets software breakpoint at line 5 (break after 100 executions) Note: Software breakpoints affect only the Q and T commands
RB (Remove Break)	RB	Removes software breakpoint at current line (if one exists)
	RB5	Removes software breakpoint at line 5 (if one exists)
	RB5,8	Removes all software breakpoints in lines 5 to 8
VB (Verify Breaks)	VB	Shows all software breakpoints
VP (Verify Ports)	VP	Shows current values of PORT A, PORT B, DDRA, and DDRB
A (Address)	A	Shows starting memory address where code for current line is stored
	A5	Shows starting memory address where code for line 5 is stored
	A5,8	Shows starting memory addresses where codes for line 5 to 8 are stored
S (Source)	S	Shows line whose code starts at current memory location
	S1C40	Shows line whose code starts at

Table 3 (continued)

COMMANDS	FORMS	DESCRIPTION
S (Source) (continued)	S1C40 (continued)	memory location 1C40
	S1C40,1C90	Shows all lines whose code starts in memory locations 1C40 to 1C90
D (Disassemble)	D	Disassembles current memory location
	D1C40	Disassembles memory location 1C40
	D1C40,1C90	Disassembles memory locations 1C40 to 1C90 Note: Disassembly does not produce original source lines
L (Load)	L	Loads an existing source file and its corresponding object file (Loader prompts for source file name)
	L TEST	Loads source file TEST.SRC and its corresponding object file TEST.OBJ
	L T.OLD	Loads source file T.OLD and its corresponding object file T.OBJ
N	N	Shows number of statements executed and current statement limit
(Number)	N500	Shows number of statements executed and sets statement limit at 500 Note: The default statement limit is 1000
Z (Zero)	Z	Zeros out all registers and counters except SP register which is set to FF (resets the simulator)
\ (backslash)	\	Prints previous line, memory location etc. (continues previous command in reverse direction, if permitted)
cr (Carriage return)	cr	Prints next line, memory location etc. (continues previous command in a forward direction)
E (Exit)	E	Exits 6502 simulator
DUMP	DUMP	Prints last 10 source instructions executed and last 10 branch instructions executed

Table 3 (continued)

COMMAND	FORMS	DESCRIPTION
K	K	Prints current line with execution count
(print count)	K5	Prints line 5 with execution count
	K5,8	Prints lines 5 to 8 with execution counts

SIMULATOR NOTES

1. The simulator permits special symbols to denote the first line or the first memory location (^), the current line or the current memory location (.), and the last line or the last memory location (*). Illustrated below are some examples using these special symbols:

P^,*	Prints all source lines
M.,*	Prints all memory locations from current location to location FFFF
RB^,*	Removes all software breakpoints
FA9,^	Fills memory location 0000 with byte A9

2. CONTROL Y is used to simulate a maskable interrup during execution of source code. Otherwise CONTROL Y functions in its usual manner on the VAX.

3. CONTROL C is used to terminate execution of source code. Otherwise CONTROL C functions in its usual manner on the VAX.

4. The hunt command remembers its last search string. Typing H without any parameters will repeat the previous hunt command but at the new location.

APPENDIX B

6502 ASSIST ERROR MESSAGES

The following is a listing of all error messages produced by the 6502 ASSIST Assembler:

Argument not in range 00:FF - value 0 assumed

Argument not in range 0000:FFFF - value 0 assumed

Address not in range 00:FF - value 0 assumed

Address not in range 0000:FFFF - value 0 assumed

Address not in range 0000:FFFF - statement ignored

Right delimiter missing - text ignored

Operand not in range -256:+255 - value 0 assumed

Extraneous symbols following right parentheses

X register can not be used for post-indexing

Y register can not be used for pre-indexing

Offset not in range -128:+127 - value 0 assumed

No grouping with parentheses - value 0 assumed

Not a valid operator - term ignored

cccccc is an undefined symbol - value 0 assumed

cccccc has an invalid digit - value 0 assumed

Only a single character permitted after apostrophe

Address field contains embedded blanks

Label must begin with a letter

Label cannot be the letter "A" or "a"

Label must contain only letters and digits

Duplicate label

cccccc is an invalid opcode

where cccccc is a label, number, or opcode mnemonic appearing in the user's source code.

The following is a listing of all error messages produced by the 6502 ASSIST Simulator:

Illegal command

Illegal syntax

Accumulator has invalid BCD digits

Memory location nnnn has invalid BCD digits

nnnn contains invalid hex digit

Source file not found

Object file not found

Code for line nnnn over-written by code for line nnnn

Insufficient space to load entire file

Software breaks not permitted at ORG/EQU pseudo-ops

Invalid opcode encountered at location nnnn

Nonsource instruction encountered at location nnnn

Invalid input value - please re-enter

BCD error executing ccc instruction at line nnnn

Invalid hex digits in filler byte

Filler value not in range 00:FF

Filler value not in range 0000:FFFF

Filler value must be 0 or 1

Invalid status bit name

Character string delimiter missing

Empty search string not permitted

Invalid hex digits in byte

where nnnn is a line or a memory address being referenced and ccc is an opcode mnemonic.

APPENDIX C

6502 ASSIST MODULES

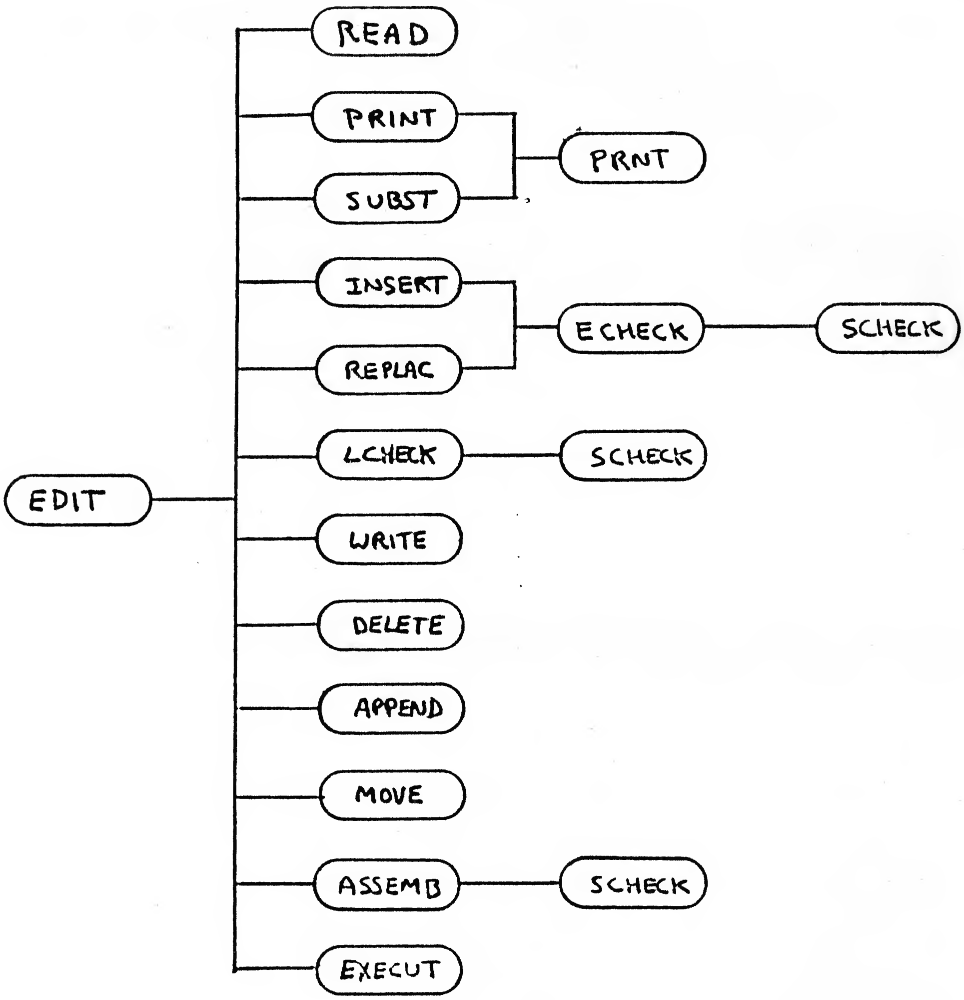

Figure 3. Editor Modules

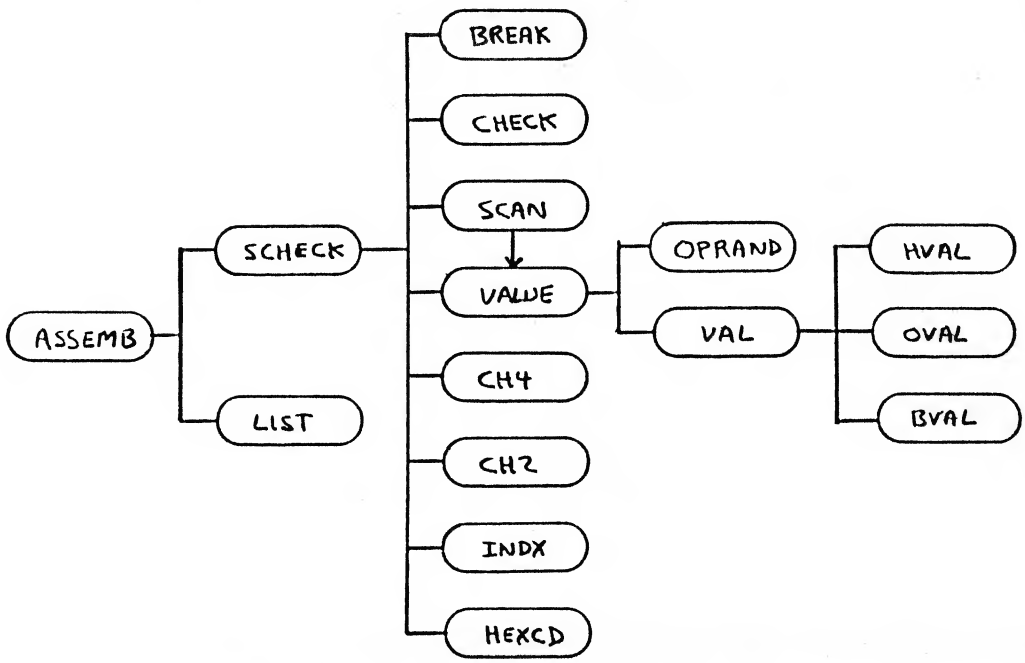

Figure 4. Assembler Modules

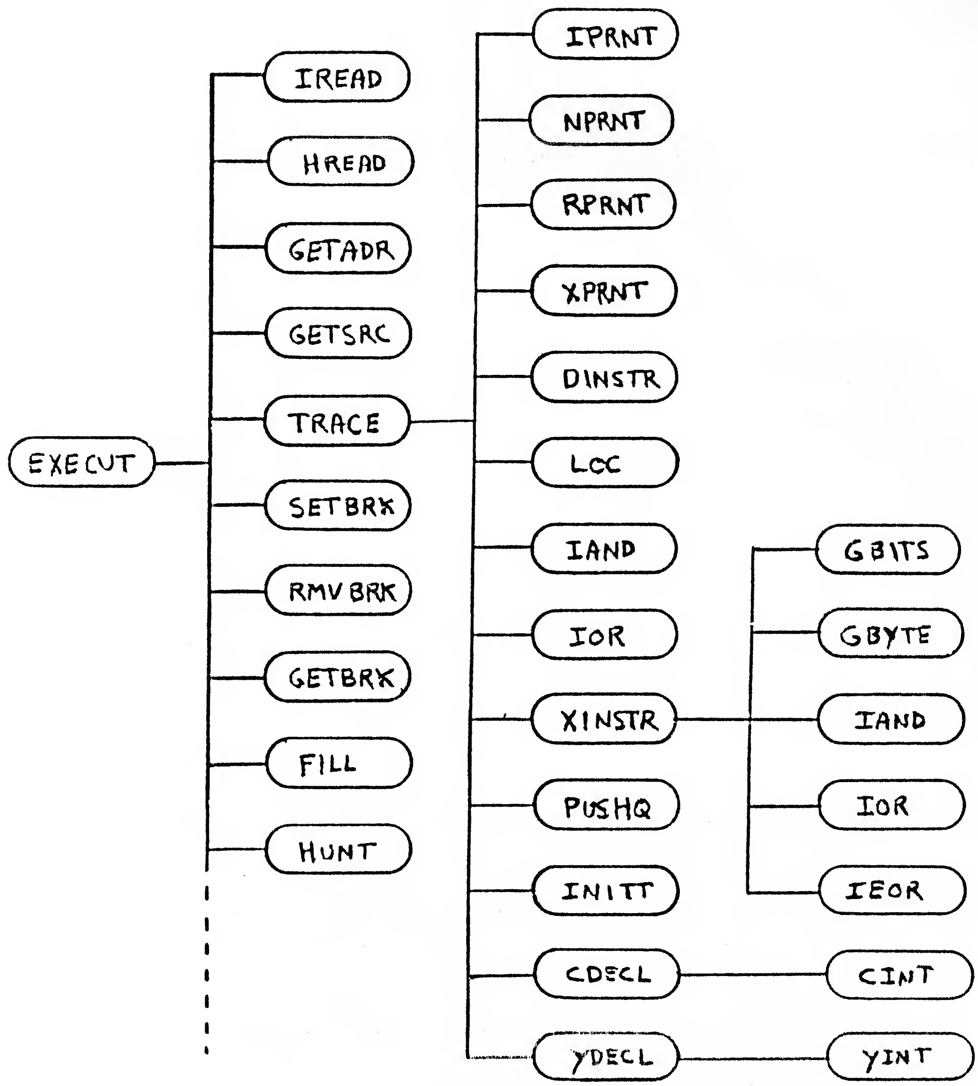

Figure 5. Simulator Modules

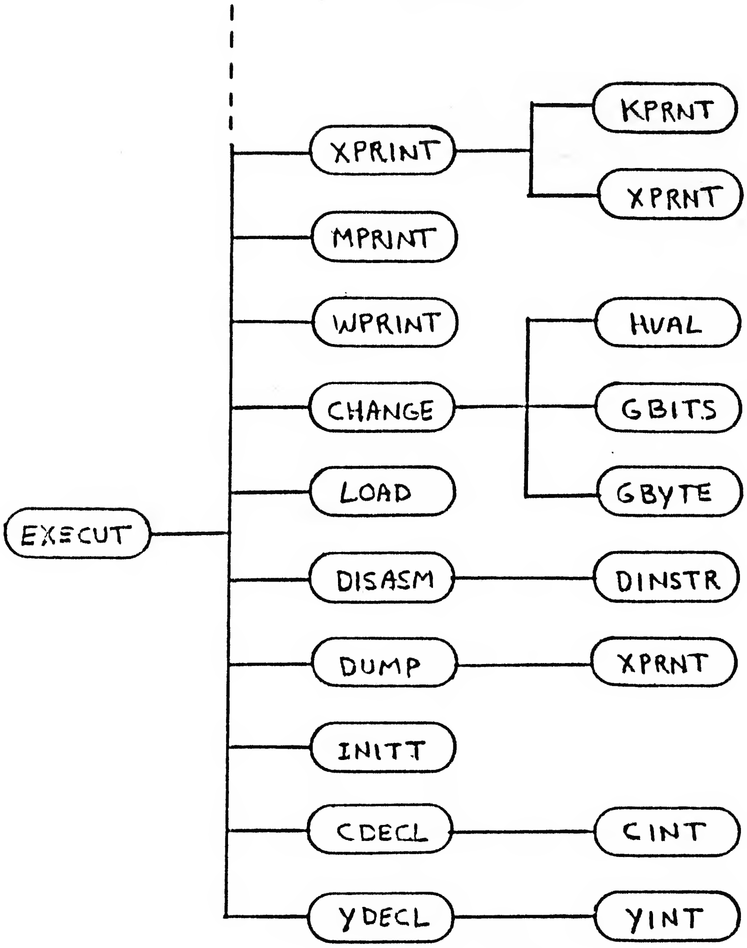

Figure 6. Simulator Modules (continued)